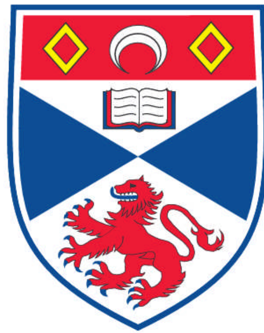
LOAD BALANCING OF IRREGULAR PARALLEL APPLICATIONS ON HETEROGENEOUS COMPUTING ENVIRONMENTS

Vladimir Janjic

A Thesis Submitted for the Degree of PhD
at the
University of St. Andrews

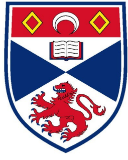


2012

Full metadata for this item is available in
Research@StAndrews:FullText
at:
http://research-repository.st-andrews.ac.uk/

Please use this identifier to cite or link to this item:
http://hdl.handle.net/10023/2540

This item is protected by original copyright



Load Balancing of Irregular Parallel Applications on Heterogeneous Computing Environments

by Vladimir Janjic

A thesis submitted to the
University of St. Andrews
for the degree of
Doctor of Philosophy

School of Computer Science
University of St. Andrews
16 December 2011

I, Vladimir Janjic, hereby certify that this thesis, which is approximately 80000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

I was admitted as a research student in October, 2007 and as a candidate for the degree of Doctor of Philosophy in September, 2008; the higher study for which this is a record was carried out in the University of St Andrews between 2007 and 2011.

Date .... Signature of candidate ....

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date .... Signature of supervisor ....

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the electronic publication of this thesis:

Access to printed copy and electronic publication of thesis through the University of St Andrews.

Date .... Signature of candidate ....

Signature of supervisor ....

Abstract

Large-scale heterogeneous distributed computing environments (such as Computational Grids and Clouds) offer the promise of access to a vast amount of computing resources at a relatively low cost. In order to ease the application development and deployment on such complex environments, high-level parallel programming languages exist that need to be supported by sophisticated runtime systems. One of the main problems that these runtime systems need to address is dynamic load balancing that ensures that no resources in the environment are underutilised or overloaded with work.

This thesis deals with the problem of obtaining good speedups for irregular applications on heterogeneous distributed computing environments. It focuses on work-stealing techniques that can be used for load balancing during the execution of irregular applications. It specifically addresses two problems that arise during work-stealing: where thieves should look for work during the application execution and how victims should respond to steal attempts.

In particular, we describe and implement a new Feudal Stealing algoritm and also we describe and implement new granularity-driven task selection policies in the SCALES simulator, which is a work-stealing simulator developed for this thesis. In addition, we present the comprehensive evaluation of the Feudal Stealing algorithm and the granularity-driven task selection policies using the simulations of a large class of regular and irregular parallel applications on a wide range of computing environments. We show how the Feudal Stealing algorithm and the granularity-driven task selection policies bring significant improvements in speedups of irregular applications, compared to the state-of-the-art work-stealing algoithms. Furthermore, we also present the implementation of the task selection policies in the Grid-GUM runtime system [AZ06] for Glasgow Parallel Haskell (GpH) [THLPJ98], in addition to the implementation in SCALES, and we also present the evaluation of this implementation on a large set of synthetic applications.

Acknowledgments

First and foremost, I would like to thank my supervisor, Kevin Hammond, for his guidance through the whole process of my PhD studies. He was truly like an academic father to me, and he provided a great help in turning my half-baked ideas into quality research. I am forever indebted to him for all that he has done for me in my four years.

Special thanks go to Steve Linton who, together with Kevin, secured the funding for my research. Without him, I would have never been able to come to St Andrews and do what I have done.

Many people from the School of Computer Science at the University of St Andrews provided help at various points in my studies. Philip Hölzenspies, Christopher Brown, Edwin Brady and Max Neunhöffer read various parts of the earlier versions of the thesis and gave very useful suggestions on how to improve it. Thanks to Alexander Konovalov and Reimer Behredens who were, together with Christopher Brown, my officemates. We always had that necessary balance between the serious and lighthearted air in our office, which created a great environment for productive work. I would also like to thank the members of Dependable System Group at the Heriot-Watt University, especially Hans-Wolfgang Loidl and Philip W. Trinder, for very useful discussions and feedback they gave me at various points for my research.

It would be unfair not to mention the support I had from various people from the Faculty of Science, Unversity of Banja Luka. I would especially like to thank the dean of the faculty, Rajko Gnjato, for ensuring financial support for my studies. I would also like to thank my undergraduate supervisor, Nenad Mitic from the Faculty of Mathematics, University of Belgrade, for introducing me to the area of functional programming.

Finally, I would like to thank my wife, Rada, for great patience and support she provided during these four years. Without her, I would have never been able to finish the thesis, and she deserves the better half of the credits for its successful completion. Thanks also to our toddler daughter Marija for her insightful babble during the long evenings of work. Also, many thanks to my father Milan, mother Spomenka and

brother Aleksandar for the continious support they gave me.

This research is supported by European Union Framework 6 grant RII3-CT-2005-026133 SCIEnce: Symbolic Computing Infrastructure in Europe.

Contents

1 Introduction **1**

1.1 Why Parallel Programming? . . . 1
1.2 Load Balancing and Work-Stealing in Parallel Runtime Systems . . . 4
1.3 Irregular Parallel Applications . . . 5
1.4 Heterogeneous Distributed Computing Environments . . . 7
1.5 Aim of the Thesis . . . 8
1.6 Contributions . . . 10
1.7 Thesis structure . . . 12
1.8 Publications . . . 14

2 Scheduling and Load-Balancing **17**

2.1 Distributed Computing Environments . . . 17
2.1.1 Computational Grids . . . 19
2.1.2 Cloud Computing . . . 22
2.2 Scheduling on Distributed Computing Environments . . . 26
2.2.1 Scheduling of Bag-of-tasks Applications . . . 27
2.2.2 Scheduling of Applications with Task Dependencies . . . 29
2.3 Load Balancing . . . 33
2.3.1 Work Stealing . . . 34
2.4 (Parallel) Functional Programming . . . 39
2.4.1 Glasgow Parallel Haskell . . . 41
2.4.2 GUM . . . 41
2.4.3 Grid-GUM . . . 45

3 SCALES Work-Stealing Simulator **51**

3.1 Overview of SCALES . . . 51
3.2 Applications . . . 53
3.3 Computing Environments . . . 56

3.4 Execution of Applications under SCALES 58
3.4.1 Accuracy of Simulations Under SCALES 62
3.5 Grid and Cloud Simulators 64
3.6 Summary 65

4 Work Stealing on Distributed Systems 67
4.1 Introduction 67
4.2 Parallel Applications 68
4.2.1 The Degree of Irregularity of Parallel Applications 74
4.3 Heterogeneous Distributed Computing Environments and Runtime systems 82
4.4 Work-stealing on Heterogeneous Computing Environments 83
4.4.1 How to Choose Steal Targets 86
4.4.2 How to Respond to Steal Attempt 89
4.5 Summary 90

5 Load-Based Topology-Aware Stealing 93
5.1 The Use of Load Information 94
5.1.1 Load-based Work-stealing Algorithms 95
5.2 Evaluation of Load-based Work-stealing Algorithms 101
5.2.1 SimpleDC Applications 105
5.2.2 The DCFixedPar Applications 122
5.2.3 Summary of Experiments 134
5.3 Feudal Stealing 136
5.3.1 The Feudal Stealing Algorithm 142
5.4 The Evaluation of Feudal Stealing 148
5.4.1 The SimpleDC Applications 150
5.4.2 The DCFixedPar Applications 152
5.4.3 Why is Feudal Stealing Better than CRS and Grid-GUM? . . . 158
5.4.4 Summary 159
5.5 Conclusions 161

6 Granularity-Driven Work Stealing 165
6.1 Introduction 165
6.2 Granularity-Driven Task Selection Policies 167
6.3 Simulations Experiments 171

6.3.1 Overview 171
6.3.2 Applications with Variable Mean Task Size 175
6.3.3 Applications with Variable Number of Tasks 183
6.3.4 Applications with a Varying Degree of Irregularity 197
6.3.5 Computing Environments with a Hierarchy of Latencies 201
6.3.6 Applications with nested-parallel tasks 208
6.3.7 Where do the Improvements Come From? 214
6.3.8 Summary of the Simulations Experiments 215
6.4 Grid-GUM Implementation 218
6.4.1 Implementation Details 218
6.5 Experiments with Grid-GUM 219
6.5.1 Differences in Simulation and Grid-GUM Setup 220
6.5.2 Experiments with the Synthetic Applications 221
6.6 Conclusions 228

7 Conclusions **231**
7.1 Contributions of the Thesis 233
7.2 Limitations of our Approach 236
7.3 Further Work 237

Chapter 1

Introduction

This thesis deals with the important topic of improving the performance of *irregular* parallel applications under *work-stealing* load-balancing algorithms on *heterogeneous distributed computing environments*. Large distributed computing environments, such as Computational Grids, offer the promise of access to a vast amount of computing power at relatively low cost. In order to ease application development and deployment on such complex environments, new languages are needed that offer high-level parallel programming constructs, while hiding most of the details of creation and management of actual parallel tasks in applications. These languages need to be supported by automatic mechanisms that distribute parallel work to the available computing resources, in order to obtain good speedups for parallel applications. The thesis considers how to develop new load-balancing algorithms (based on work-stealing mechanism) that can give improved runtime performance, especially for the important, but rarely studied, class of symbolic computing applications. The work has been tested using both a novel simulator, and by adapting the Grid-GUM runtime system, a distributed implementation of the parallel functional language Glasgow Parallel Haskell (GpH).

1.1 Why Parallel Programming?

In the last 10 years or so, we have witnessed a real revolution in the way in which computer applications are written. Since the beginnings of computing, the hunger of applications for more processing power has constantly increased. For many years, single-processor systems were able to improve in line with increasing applications' demands, but this is not the case any more. We can see nowadays that the clock speeds of modern processors are improving very slowly, or, in the case of some architectures, have completely stalled or are even reduced. In order to provide more cycles for ever-

demanding applications, computing systems are becoming more and more *parallel*. This is not the case just for high-end specialised systems (such as supercomputers). Nowadays, even systems intended for home usage typically come in the form of dual- or quad-core processors. We expect the trend of increasing parallelism in computing architectures will continue for the foreseeable future, so that future systems will not only be *slightly* parallel , they will be *massively* parallel.

All of this has had an important impact on application developers. It is no longer possible to rely on the computing architecture alone to provide the necessary cycles to ensure improved performance for demanding sequential applications. Consequently, we have seen a big shift from *sequential* to *parallel* programming paradigms. However, transforming legacy sequential applications into parallel ones which give good speedups on modern architectures is rarely an easy task, since good sequential applications often employ elaborate optimisations that usually make them inherently sequential and, therefore, difficult to parallelise. Frequently it is easier to write a parallel application from scratch, than to transform an already existing sequential one. Therefore, in order for applications to perform well on a range of current and future architectures, programmers nowadays *must think in parallel* from the very early stages of an application's development.

Writing parallel applications, however, can be a really daunting task, especially when it is done in low level systems (such as C+MPI [Qui03]). There are many more things that can go wrong when writing parallel applications than when writing sequential applications. For example, parallel application may introduce race conditions, which can result in an inconsistent application behaviour or in deadlocks. Even when we manage to produce a correct parallel application, its performance on a parallel machine can be worse than that of an equivalent sequential program, for example when task sizes are too small or there are too many data dependencies between them. Therefore, many programming languages (or the extensions to already existing languages) which simplify the writing of parallel applications have been developed. In these systems, a programmer usually has a small set of parallel primitives which can be used to denote functions (or expressions) that can be executed (or evaluated) in parallel. Examples of such languages include:

- *Cilk* [BJK+95], an extension to the programming language C, which includes the constructs for declaring and spawning parallel threads and constructs for inter-thread communication.

- *Javelin* [NC02], a software system based on Java, designed for the development of large-scale distributed parallel applications. Javelin includes a large number

of constructs, which support common forms of parallel programming, such as Single-Program-Multiple-Data (SPMD).

- *GpH* [THLPJ98], an extensions to the programming language Haskell [Jon03], which includes `par` and `seq` constructs. The `par` construct denotes the two expressions which should be evaluated by parallel threads, whereas `seq` orders the evaluation of expressions.

These parallel languages are designed to target general parallel applications. An alternative is to use parallel environments that further ease parallel programming by offering even higher levels of abstraction, by hiding more of the low-level details of thread creation and synchronisation from the programmer. The penalty paid in these environments is that they usually restrict the kind of applications that can be developed to those conforming to specific parallel paradigms. Examples of such environments include:

- *Satin* [VNWJB10], a programming environment for Java, which targets divide-and-conquer and master-worker parallel applications.

- *Single Assignment C (SAC)* [GS06], a functional programming language, which focuses on applications based on parallel operations on arrays.

- *eSkel* [MCGH05], a parallel library, based on C and the MPI [GLS99] communication library, which supports parallel programming using algorithmic skeletons [Col89]. Algorithmic skeletons represent the abstraction of commonly-used patterns of parallel computations, communication and interaction.

Making parallel applications easier to write, however, does not come for free. Parallel languages and environments, that offer a high level of abstraction, usually rely on very sophisticated runtime systems to do most of the hard work related to the parallel execution of applications, such as creation of parallel tasks and their communication, synchronisation and scheduling. The mechanisms that such runtime systems employ for task scheduling, that involve decisions about what tasks will be run where, are probably the most important to ensure good performance of parallel applications. In this thesis, we focus on one particular aspect of task scheduling – that of *load-balancing* decisions. Load-balancing methods are responsible for ensuring that the work is evenly distributed across all nodes of a parallel machine during the application execution.

1.2 Load Balancing and Work-Stealing in Parallel Runtime Systems

One of the most crucial things that developers of runtime systems need to consider is how load balancing is done. In the context of runtime systems for parallel applications, load balancing refers to transferring the parallel tasks between different nodes of a parallel machine, in order to ensure that each node has approximately the same amount of work to do. If the work is not equally distributed across nodes, then some of them will be idle while others will have too much work, so we cannot hope to achieve good speedups of parallel applications.

We assume that each node holds the parallel tasks it creates in its own *task pool*. Alternatively, in shared-memory settings, there may be a common task pool to which all nodes have access. Keeping this in mind, load-balancing methods can be classified into two main classes, depending on who initiates the balancing:

- *Sender-initiated* or *Work-pushing* methods are those in which a node that has more than some predefined number of tasks in its task pool automatically distributes them to other nodes. Alternatively, if a common task pool exist, whenever a new task is added to it, it is automatically sent to some node. All load-balancing decisions, at each point in the application execution, are, therefore, initiated by nodes that have multiple tasks in their pools (or that add tasks to a common task pool).

- *Receiver-initiated* or *Work-stealing* methods are those in which nodes do not automatically offload tasks they create to other nodes. Instead, they keep them in their task pools and only when idle nodes (those that do not have any task to execute) ask them for work, do they send tasks to them. If a common task pool exist, idle nodes grab tasks from it. We can see that load-balancing decisions in these methods are initiated by nodes that need to obtain tasks.

Both classes of load balancing methods have their advantages and disadvantages. The main advantage of work-pushing is that it incurs less communication than work-stealing in distributed memory settings. In order to transfer a task from node A to node B, work-pushing just requires that node A contacts node B (e.g. by sending a message with a task to it) whereas work-stealing requires both node B to contact node A (e.g. by sending some kind of steal-request message) and node A to contact node B (e.g. by sending a message with the task to it). The main advantage of work-stealing is that overloading nodes with work is much less likely to occur. Since nodes that send

work in work-pushing do not necessarily know the load of a receiver, a situation can happen where a node that already has many tasks to execute receives additional tasks and becomes overloaded with work.

It is generally the case that work-stealing methods give a better overall load balance, and the penalty paid in more communication is not that significant (see, for example, the comparison between work-stealing and work-pushing algorithms in Van Nieuwpoort et al. [VNKB01]). Because of this, many runtime systems (e.g. Cilk, Satin, Javelin, GpH) use work-stealing as a method for load balancing. In this thesis, we will, therefore, assume that load balancing is done using work-stealing.

In the rest of the thesis, we will adopt the following terminology for work-stealing in the settings where each node has its own task pool. An idle node, which needs to obtain tasks, will be called a *thief*. A node, that is asked for work by a thief, will be called a *target*. Note that in work-stealing it is not certain whether a target node will have tasks to send to the thief. A target that has tasks in its task pool and that, therefore, can send some tasks to the thief will be called a *victim*. A message in which a thief asks a target for work will be called a *steal attempt*. A *steal operation* consists of a steal attempt made to a victim, together with the message(s) where the task is sent as a response.

1.3 Irregular Parallel Applications

The main focus of this thesis is on *irregular* parallel applications. Generally, there is no universally accepted definition of what an *irregular application* is, i.e. what is it that distinguishes *regular* from *irregular* applications. However, irregularity in the literature usually refers to one of two different concepts. The first one takes into account the high-level structure of an application in terms of, for example, the data structures used (vectors and matrices for regular applications, and pointer-based data structures, such as sparse graphs, for irregular ones, e.g. Kulkarni et al. [KPW+07], Belloch et al. [BHC+93]) or algorithms or parallel paradigms used (e.g. referring to applications with a single level of parallelism as regular, and those with nested parallelism as irregular). We will call this kind of irregularity the *structural irregularity* of an application. The second concept takes into the account the actual tasks that are created during the execution of the application, seeing an application as irregular if different tasks either: i) have different sizes; ii) generate different amounts of parallelism; or iii) have different communication patterns (e.g. Barker and Chrisochoides [BC03], Nikolopulos et al. [NPA01]). We will call this kind of irregularity the *cost irregularity* of an ap-

plication. We can see that cost irregularity focuses on lower-level issues (that is, the level of actual parallel tasks created at execution time) than structural irregularity.

It is almost always the case that structurally irregular applications are also cost irregular, but not the other way around. A lot of applications that have simple, regular high-level structure create tasks with cost irregular parallelism. For example, consider the parallel version of the `sumEuler` function written in GpH:

```
sumEuler l = sum (parMap rnf euler l)
```

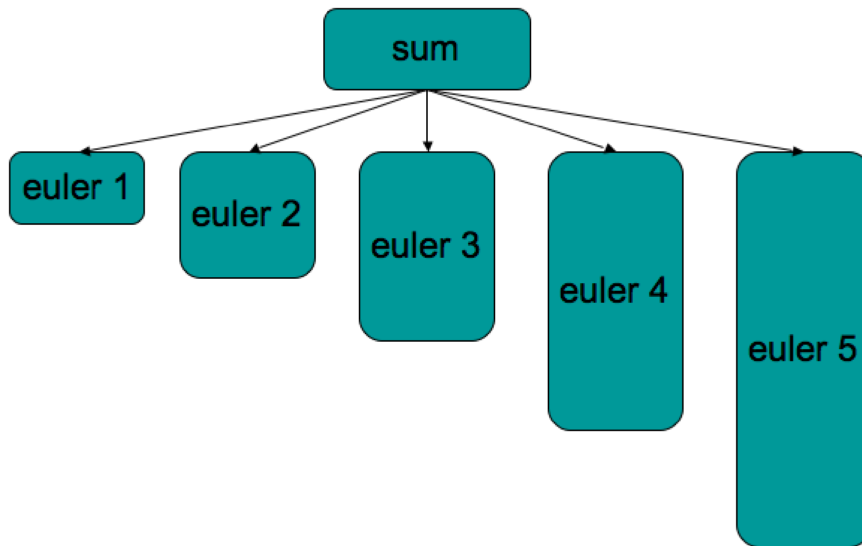


Figure 1.1: sumEuler application

This function maps the `euler` totient function over the list of integers in argument l and then calculates the sum of the resulting list. Its high-level structure is that of a single-level data parallel application (applying sequentially the same operation over the list of numbers in parallel), which makes it structurally regular (Figure 1.1). However, since the cost of applying the `euler` function is different for different input values, the actual parallel tasks created during the application execution have different sequential sizes, which makes it cost irregular.

In this thesis, we will focus solely on cost irregular applications. Therefore, when we say "irregular" we will mean "cost irregular". The main motivation for that is that cost irregularity deals with lower level issues, which are more "measurable". Rather than just distinguishing between "regular" and "irregular" applications, we want to be able to tell precisely *how irregular* the application is.

The main motivation for considering irregular applications comes from the area of symbolic computations, which is targeted by the ongoing SCIEnce research project [AZTH+08]. Symbolic computations deal with finding the exact solutions to mathematical problems which involve symbolic objects, as opposed to operating with approximations on numerical values in numeric computations. These computations are typically time and/or memory consuming, and are based on irregular high-level algebraic data structures with highly irregular parallelism (see the examples of symbolic computations in Linton et al. [LHK+11] and Al Zain et al. [AZTH+08]).

1.4 Heterogeneous Distributed Computing Environments

At several points in the previous discussion, we have mentioned the concept of a parallel machine. Rather than a physical machine, our concept of a parallel machine is more a logical one. A parallel machine may involve a large number of geographically distributed high-performance computing servers, connected by high-latency networks, that have the appropriate software infrastructure that enables parallel applications to see them as a single machine.

The parallel machines (or *computing environments*, as we will call them in the rest of the thesis) that we consider consist of a set of nodes (processing elements or PEs) with their associated memories plus the networks by which they are connected. We assume that each PE has its own private memory, and that there is no memory that is shared between PEs. The only means of communication between PEs is through the exchange of messages via networks. These kinds of environments are usually called *distributed* or *message passing* systems. Again, we emphasise that our concept of a computing environment is a logical one, as it describes how the runtime systems see the underlying physical hardware, rather than what this hardware is. It is perfectly possible to implement distributed computing infrastructures on top of a physical shared-memory hardware. For example, in a multicore machine, PEs may correspond to processor cores, sending a message may correspond to writing some data to a shared memory and receiving a message may correspond to reading from the shared memory.

The main focus of the thesis is on *heterogeneous* distributed computing environments. In such environments, the capabilities of individual PEs (in terms of how fast they can execute tasks), communication latencies between PEs and the amount and type of memory associated with each PE can be different. This is the typical setup of, for example, Computational Grids, which usually consist of a set of clusters connected

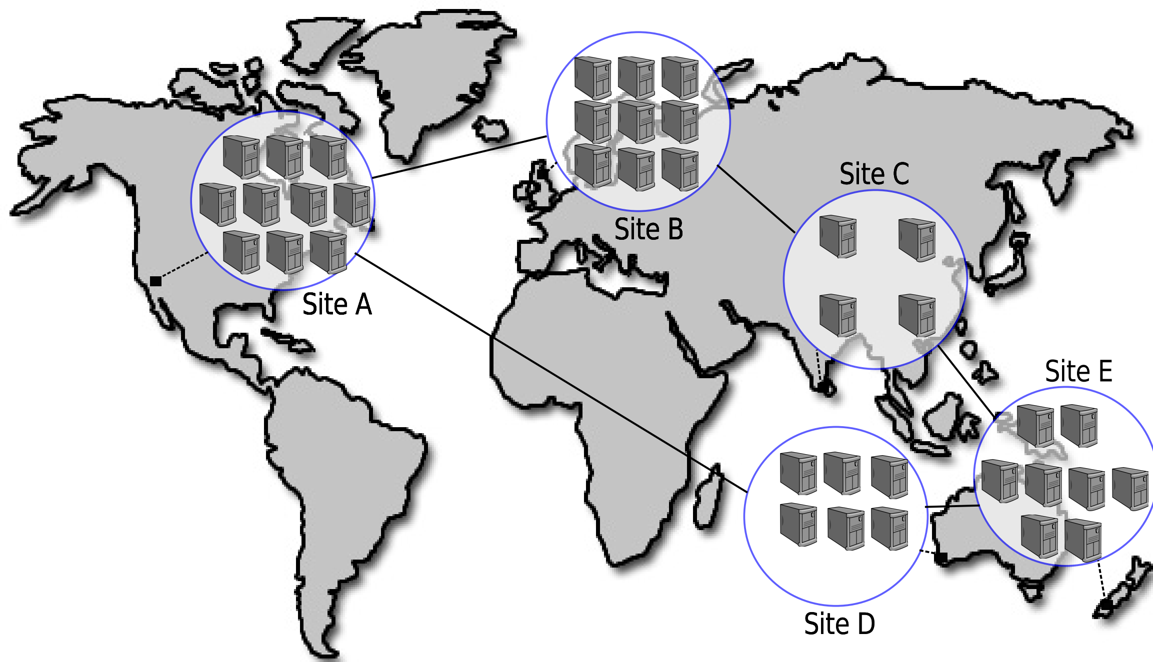


Figure 1.2: Computing environment where clusters of PEs from 5 sites from Europe, North America, Asia, Australia and New Zealand are connected by high-latency networks

by high-latency networks (see Section 2.1.1 for more details about Grids). Each cluster consists of a number of PEs. Computing powers of PEs in different clusters are usually different and the communication latency between PEs from the same cluster is usually much lower than for PEs from different clusters. See Figure 1.2 for an example computing environment with heterogeneous communication latencies between PEs.

The main motivation for considering heterogeneous distributed computing environments is that they offer much more processing potential that tightly-coupled systems.

1.5 Aim of the Thesis

Efficient work-stealing on heterogeneous distributed computing environments is hard, especially for irregular parallel applications. In such environments, sending stealing related messages can be very expensive, especially when communication latency between a thief and a target is high. Therefore, it is imperative for thieves to choose steal targets very carefully, in order not to send too many messages to the targets that do not have any work to send. At certain phases of the execution of irregular parallel applications, there may only be a small amount of parallelism available or all

available tasks may be concentrated on a small number of PEs. Locating these PEs in a potentially large environment can be very hard, and we might end up in a situation where thieves spend a lot of time locating work, which results in reduced applications' speedups.

The second problem comes from the fact that tasks of irregular applications can have different sizes and they can generate a different amount of additional tasks. We need to be careful in deciding which tasks a victim should send to a thief, in order to preserve the overall load balance. Sending too small tasks over high latency networks can result in situations where communication time exceeds execution time. On the other hand, we also need to be careful not to send too many large tasks or tasks that generate a lot of parallelism on the same group of PEs, otherwise they may get overloaded with work.

Further complications for work-stealing in irregular applications can come from different communication patterns of different tasks, so we must be careful not to place the tasks with a lot of data dependencies on PEs which communicate over high latency networks, as the overheads in communication between these tasks can be too high.

The current state-of-the-art work-stealing algorithms do not address the problems mentioned above in a proper way. Most of the work-stealing algorithms currently used in runtime systems are tailored to the execution of the applications of a simple, regular structure (such as divide-and-conquer parallel applications where every task, up to some threshold, generates further tasks, or data-parallel applications where all of the tasks are of approximately the same size). The other problem is that they do not adapt very well to highly heterogeneous computing environments, especially when a whole hierarchy of communication latencies between different PEs exists.

In this thesis, we aim to improve the current state-of-the-art work-stealing algorithms to deliver better speedups for irregular parallel applications on heterogeneous distributed environments. It, therefore, represents the first attempt to develop work-stealing algorithms that will be especially tailored to irregular parallel applications. Specifically, we target two main questions that arise during work-stealing:

- How should thieves choose targets for their steal attempts?
- How should victims respond to these steal attempts, i.e. what tasks should they send as a response?

In order to answer the first question, we present a novel Feudal work-stealing algorithm, which uses the information about dynamic PE loads (where by PE load we mean the number of tasks in its task pool) in making decisions about where thieves

should send their steal attempts. Our algorithm is novel in that it uses a combination of locally-centralised and remotely-distributed methods for propagation of the information about PE loads, informed by the communication hierarchy in the underlying computing environment. We show that by using this method we can obtain a very good approximation of dynamic PE loads, and that we can significantly improve speedups of irregular parallel applications compared to algorithms that don't use load information, or those that use fully distributed methods of information propagation.

To answer the second question, we focus on the information about the parallel profiles of tasks that comprise parallel applications being executed. In particular, we investigate how the information about task sizes can be used to improve work-stealing. We propose different granularity-driven task selection policies that victims can use to select the tasks that should be sent as responses to steal attempts. We show that these policies can notably improve applications' speedups compared to ad-hoc policies currently used in work-stealing algorithms, which choose the tasks based on their age.

All of our algorithms are evaluated using a highly-parametrised simulator of work-stealing on distributed systems (developed for this thesis). Furthermore, we describe the implementation in the Grid-GUM Parallel Runtime System [AZ06] for Glasgow Parallel Haskell of the granularity-driven task selection policies. These experiments show both theoretical improvements in speedups that our methods bring, compared to the state-of-the-art work-stealing algorithms used in runtime systems, as well as their practical applicability in real systems.

1.6 Contributions

The main contributions of this thesis are

- *A novel* Feudal Work-Stealing *algorithm for work stealing on heterogeneous computing environment*
  Our algorithm uses information about the network topology of the underlying computing environment, together with a combination of locally-centralised and remotely-distributed methods for propagation of information about relative PE loads to guide the decisions of *where* should PEs send steal requests. We show that our algorithm outperforms the state-of-the-art work-stealing algorithms for heterogeneous computing environments, and also that the relative PE load information that can be obtained with it is more accurate than under state-of-the-art Grid-GUM work-stealing algorithm, which uses a fully distributed method of load information propagation.

- *Novel policies for selecting the tasks for offloading during the work-stealing*
  We propose policies for selecting task(s) that will be sent as a response to steal requests made from thieves during work-stealing. Our tasks are novel in that they use application-specific information about task sizes in making their decisions. Since they deal with the question of *how to respond* to steal requests, rather than *where to send* them (which is the focus of most of the work stealing algorithms), they are orthogonal to most of the work stealing algorithms and can, therefore, be readily "plugged" into all of them, if the required information about task sizes is available.

- *Comprehensive evaluation of all of our work-stealing algorithms and policies using simulations*
  We have done a comprehensive evaluation of the algorithms proposed in this thesis on a wide class of parallel applications and computing environments using simulations. This has enabled us to make a conclusions about the situations (i.e. application/computing environment combinations) for which the improvements in speedup under our algorithms are the best (compared to other state-of-the-art algorithms).

- *Implementation of granularity-driven task selection policies in Grid-GUM*
  We describe the implementation of granularity-driven task selection policies in the Grid-GUM runtime environment, and the evaluation of performance of this implementation for a wide class of parallel applications. This shows both the practical applicability of the proposed task selection policies and the accuracy of our simulations, since the results we obtained for the same classes of applications under simulations and real implementation match.

- *Analysis of the usability of load information in state-of-the-art work-stealing algorithms*
  We analyse how much the state-of-the-art work-stealing algorithms could benefit if they have *perfect* information about the dynamic loads of all PEs during the whole application execution. In other words, we investigate how much better would they perform if a thief were to know the load of every other PE in a system each time it needs to make work-stealing decisions. This enables us to conclude what is the best way of using the load information in work-stealing, and also it gives important information to the implementers of work-stealing runtime systems on how much their algorithms would benefit if they were extended with the usage of load information.

Additionally, the thesis also makes the following minor contributions:

- *The precise definition of the degree of irregularity of an application (with respect to task granularity)*
  Whereas in the literature parallel applications are classified simply as either regular or irregular, we have developed the precise mathematical definition of the *degree of irregularity* of a parallel application. This tells us exactly 'how irregular' the application is, and what is the impact that application irregularity has on the performance of both currently used work-stealing algorithms, and also the Feudal work-stealing algorithm. Using this definition, we show the very important fact that the algorithms and policies that we propose in this thesis bring better improvements in speedups (compared to the other state-of-the-art work-stealing algorithms) for *more irregular* parallel applications.

- *Highly-parametrised simulator of work-stealing*
  We have developed the SCALES simulator for work stealing on distributed computing environments, which can simulate the execution of a wide class of parallel applications on different heterogeneous computing environments. SCALES enables us to evaluate our work-stealing algorithms on a much wider class of applications and computing environments than if we were to use only the implementation of these work-stealing algorithms in runtime systems.

1.7 Thesis structure

The structure of this thesis is as follows:

- **Chapter 2** gives a general survey of heterogeneous computing environments, focusing on Computational Grids and Clouds. We also give a survey of different approaches that are currently used in scheduling/load balancing both on homogeneous and heterogeneous computing environments. We pay special attention to the approaches that most similar to ours either in that they use work-stealing or that they consider similar types of applications (e.g. workflow scheduling algorithms). Additionally, this chapter gives a brief overview of parallel functional programming, focusing on Glasgow Parallel Haskell (GpH) and the GUM parallel runitme system for GpH on distributed environments. We also describe the extension to GUM for Computational Grids, Grid-GUM, which represents the work that is most closely related to ours. Grid-GUM serves us as a testbed

for the implementation of the granularity-driven task selection policies that we consider in this thesis.

- **Chapter 3** describes the SCALES work-stealing simulator for heterogeneous distributed computing environments. SCALES was developed for the purpose of testing the work-stealing methods considered in this thesis, and is highly parametrizable. It enables us to simulate a wide class of parallel applications on a wide class of computing environments, and also allows us to tune the parameters of the simulated runtime system, enabling simulation of different runtime systems. We also give a brief overview of other scheduling simulators, focusing on Grid/Cloud simulators.

- **Chapter 4** describes, in more detail, the approach that we take in improving the state-of-the-art of work-stealing algorithms, by describing in more detail the kind of applications that we consider and the restrictions on computing environments and runtime systems we impose. We also define an important concept of the *degree of application's irregularity*, which we heavily use in subsequent chapters. We subsequently describe in more detail the problems that irregularity of parallel applications pose to work-stealing on heterogeneous distributed environments, and describe our approach to solving them. As we have mentioned in Section 1.5, we focus on two main questions, that of *locating* the parallel tasks, and that of *choosing a task* to send as a response to steal attempt.

- **Chapter 5** deals with the first of two questions we have posed. In this chapter, we investigate how both static (communication latencies between PEs) and dynamic (PE loads) information about the computing environment on which parallel applications are executed can be used to improve the work-stealing algorithms. After showing *how to use* this information, we show *how to obtain* it. In particular, we present the *Feudal Work-Stealing* algorithm, which can be used for the purpose of PEs obtaining good approximation about the dynamic system load at each point in the application execution.

- **Chapter 6** investigates the second question that we have posed. We show that by using information about task granularity, we can make better decisions about which tasks to offload as a response to work-stealing requests than if this information is not present. We propose different granularity-driven task selection policies and evaluate in which situations each of them works well. Furthermore, we show that the amount of improvement we obtain when using task granularity

information is directly related to the degree of irregularity of parallel application that is being considered, and that the more irregular the application is, the better are the improvements that we can get.

- **Chapter 7** concludes the thesis, by giving an overview of what has been achieved in the thesis, reviews its contributions and points out its limitations and the ways in which work done here could be further extended.

1.8 Publications

During the work on this thesis, we have published the following papers:

1. Vladimir Janjic and Kevin Hammond. Granularity-Aware Work-Stealing for Computationally-Uniform Grids. In *Proc. of 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGrid 2010, pages 123–134. IEEE Computer Society, May 2010

   Most of the material from Chapter 6 is based on this paper, where granularity-driven task selection policies are proposed and evaluated. Chapter 6 presents more comprehensive evaluation of these policies, and more detailed description of their implementation in Grid-GUM. The paper, on the other hand, additionally describes the experiments with a small set of realistic GpH applications, and shows how can the granularity-driven task selection policies be used to improve their speedups.

2. Vladimir Janjic and Kevin Hammond. Prescient Scheduling of Parallel Functional Programs on the Grid. *Draf Proc. of Intl. Symposium on Trends in Functional Programming*, TFP 2008. 2008.

   In this paper, we presented the design of a dynamic *prescient* scheduler, based on Grid-GUM, for irregular parallel applications on Computational Grids. It also showed how some simple information (the number of threads created at regular time intervals over the application execution) about the profile of a parallel application can be used to increase the PE utilisation.

3. Vladimir Janjic, Kevin Hammond and Yang Yang. Using Application Information to Drive Adaptive Grid Middleware Scheduling Decisions. In *Proc. of the 2nd Workshop on Middleware-application Interaction*, MAI 2008, pages 7–12. ACM, 2008

In this paper, we propose a method for obtaining information about the task sizes of an application, based on the history of previous runs of the same application. This method is based on the Markov-chain statistical model, and it tries to capture common patterns of task sizes that appear in multiple executions of the same application. This method can be used for obtaining the information about the task sizes of the applications for which task sizes depend on the particular inputs to the application.

4. Henrique Ferreiro, Victor Gulias, Kevin Hammond and Vladimir Janjic. Repeating History: Execution Replay for Parallel Haskell Programs. *Draft Proc. of 23rd Symposium on the Implementation and Application of Functional Languages*, IFL 2011. 2011. The final version of the paper is in preparation.

   This paper outlines the technique for the execution replay of parallel Haskell programs. This technique is useful in debugging parallel applications, as well as in obtaining information about the profiles of parallel tasks.

Chapter 2

Scheduling and Load-Balancing on Distributed Computing Environments

In this chapter, we give an overview of the most popular classes of large-scale distributed computing environments that are in use today, namely, Computational Grids and Clouds. After that, we describe scheduling methods used on these environments. Specifically, we focus on Grid scheduling. Although we pay special attention to the methods used for scheduling of the applications that have interdependence between tasks, we also give an overview of scheduling of more simple, bag-of-tasks applications. We subsequently focus on load-balancing methods, paying particular attention to the work-stealing algorithms used in modern runtime systems. Finally, we give an overview of parallel functional programming and, specifically, of the Grid-GUM runtime system for Glasgow Parallel Haskell (GpH), which is used as a testbed for the implementation of some of the load-balancing methods considered in this thesis.

2.1 Distributed Computing Environments

In general, the notion of a *distributed model* can refer both to the programming model used in parallel applications and to the underlying computing environment on which these applications are executed. When used to qualify the programming model, distributed model (or, as it is often called, a *distributed-memory model*) refers to an application that consists of a collection of processes, each of which has its own private address space, and where the only mean of process communication is message passing. Accessing some data by message passing is typically much more expensive

than by reading it from the private memory of a process. Applications written using C+MPI [GLS99] are typical examples of applications that conform to distributed application model.

When used to qualify the computing environment, distributed model refers to a collection of *nodes* (where nodes, in general case, can be processing elements, disks, sensors etc.) connected by networks, where each node has its own physical memory associated with it. The only mean of communication between nodes is by sending messages via networks. Typical examples of distributed computing hardware systems are Beowulf clusters [GLS03] and Computational Grids [FKT01, BFH03].

Shared-memory model refers to the applications (computing environments) that consist of a collection of threads (processors) that share the common address space (physical memory). The distinction between the shared-memory and the distributed model in computing environments is not always obvious. Take, for example, a modern multicore machine. Obviously, we can see such a machine as a shared-memory computing environment, since all of the cores have access to the shared main memory. However, since each core has its own L1 cache memory, we can also see it as a distributed computing environment, where message passing consists of one core writing the data from its cache to the main memory (or to the L2 cache memory, if this is shared between multiple cores) and the other core reading this data into its own cache. Moreover, the main optimisation objectives for applications for multicore machines are very similar to these for distributed ones, i.e. to keep the data local as much as is possible (in the local memory of a node in distributed settings, or in L1 cache on a multicore machine).

Although, usually, distributed-memory applications are implemented on a distributed computing environments (and, similarly, shared-memory applications on a shared-memory environments), other combinations are also possible. For example, the GUM [THM+96] and the Grid-GUM [AZ06] runtime environments use the concept of virtual-shared memory, which is implemented on top of distributed computing environment (see Section 2.4.3 for more details). It is also possible to run GUM and Grid-GUM on a shared-memory machine, provided that the suitable implementation of MPI communication library is available. In this way, we get a virtual-shared memory application model for distributed computing environments, which runs on physically shared-memory computing environment!

In this thesis, we will solely focus on runtime systems that conform to the distributed-memory application model. Additionally, we will be interested in running these runtime systems on distributed computing environments. As we have described in Section

1.4, we are interested in the computing environments where the only type of nodes are PEs, which are capable of executing tasks. We are especially interested in large-scale heterogeneous distributed computing environments, where computing resources from several geographically distributed sites are connected in a single unified environment or, alternatively, where a large number of virtual machines is distributed over a large data centre, consisting of many computing nodes (typical setup of Clouds).

In the next two sections, we give an overview of two important classes of large-scale heterogeneous distributed computing environments – Computational Grids and Clouds.

2.1.1 Computational Grids

The problem that underlines the Grid concept is that of *coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organisations (VOs)* [FKT01, BFH03]. Virtual organisation can be seen as a set of individuals and/or institutions that is sharing some set of resources under strict rules that define what is shared, who is allowed to share and under which conditions does the sharing occurs. Resources that are shared can range from physical (networks, sensors, disks, computers etc.) to logical ones (application, databases etc.), and can be highly heterogeneous. *Grid* itself can be seen as a software infrastructure (middlewares, APIs, SDKs, protocols, interfaces etc.) that enables this sharing to occur. This infrastructure includes security solutions that support managements of credentials and policies when computations span multiple institutions, resource management protocols and services that support secure remote access to computing and data resources and the co-allocation of multiple resources, information query protocols and services that provide configuration and status information about resources and so on [FKT01].

The term Grid is used by analogy with the electric power grid [FK04b], which provides pervasive access to electricity and has had a dramatic impact on human capabilities and society. The main vision behind Computational Grid is similar - to allow users transparent and flexible access to a vast amount of computational power and storage, while hiding most of the “gory” details about the management of underlying resources. The development of the Grids was, and most likely will continue to be, primarily driven by the requirements of large scientific applications and projects [BH04].

The Evolution of Grids

The beginnings of Grid computing can be traced back to the end of 1980s, with several projects to link different supercomputing sites [RBJ03]. Early to mid 1990s mark the emergence of Grid environments, where the main objective was to provide computational resources to a range of large high-performance applications. Representative examples of Grid projects at this time were FAFNER [FAF] and I-WAY [FGN+03]. The objective of the FAFNER (Factoring via Network-Enabled Recursion) project was to harness the spare cycles from workstations of users throughout the world in order use the Number Field Sieve (NFS) method for factoring large numbers. The purpose of the experiment was to test the security of RSA public key encryption algorithm. I-WAY (Information wide area year) was a project conceived in 1995. Idea behind it was to, using the existing high bandwidth networks, link many high-performance supercomputers and advanced visualisation environments in one large uniform system in order to allow the execution of a range of large high-performance applications. Many components developed during the I-WAY project (e.g. mechanisms for uniform authentication, resource reservation, process creation, communication, resource scheduling across different sites) served as a starting point for development of modern Grid middlewares, for example Globus toolkit [Fos03].

From the experiences obtained with these early Grid projects emerged the second phase of Grid evolution [RBJ03]. Inspired by the I-WAY infrastructure, the focus of this evolution phase was development of common middlewares for Grid computing which will better tackle the problems of hiding the heterogeneity of Grid resources, allowing Grids to scale easily to millions of resources and enabling various fault-tolerance mechanisms. The key projects that emerged from this phase were Globus and Legion [GWT97] middlewares. Of these two, Globus became de-facto standard middleware for Grid computing, on top of which most of the Grid systems used today are built. We will describe Globus in more detail later.

A very important development in the second phase of Grid evolution happened in resource scheduling area, where several schedulers and resource brokering systems which deal with resource heterogeneity were designed. The objective of most of these systems, such as Condor (and Condor-G) [TL04], PBS [PBS] and Nimrod/G [Buy02], was the execution of independent, batch jobs on non-dedicated resources.

The third phase of Grid evolution added a further level of abstraction over the middleware tools developed in the second phase. The focus there was on service-oriented model and increased usage of meta-data. Similar model was later widely adopted with the emergence of Clouds (see Section 2.1.2). The most prominent examples of

standards developed in this phase of Grid evolution are the Open Grid Services Architecture [FKT04] and World Wide Web Consortium set of protocols and standards (e.g. SOAP, Web Service Description Language, Universal Description Discovery and Integration) for describing Web services.

We can see that, through their evolution phases, Grids have evolved from very specific, commercial, ad-hoc solutions for connecting geographically distributed workstations or supercomputers on relatively small scale to much more general middlewares and standards which abstract over the resources used, offering the whole Grid infrastructure as a service.

Middleware for Supporting Large Parallel Applications on Computational Grids

In this section, we briefly describe the tools that Grid infrastructure provides for supporting the execution of large distributed parallel applications. The main components of Grid infrastructure related to this are Globus toolkit and MPICH-G2 communication library.

Globus toolkit is a community-based, open-architecture, open source set of services and software libraries that support Grids and Grid applications [FK04a]. Globus deals with the issues of security, information discovery, resource management, data management, communication, fault tolerance and portability. The most important components of the toolkit, concerned with the execution of parallel applications, are:

- *Grid Resource Allocation and Management* (GRAM) protocol provides secure and reliable creation and management of remote computations. Main parts of the Globus implementation of GRAM protocol are *gatekeeper* process, which initiates remote computations, *job manager*, which manages the remote computations and *GRAM reporter*, which monitors and publishes information about the identity and state of the local computations.

- *Monitoring and Discovery Service* (MDS) provides mechanisms for discovering, accessing and publishing configuration and status information about Grid resources, such as compute servers, networks and databases.

- *Grid-FTP* file transfer protocol, which is the extended version of the FTP protocol. Compared to the basic version of FTP protocol, Grid-FTP adds use of Grid specific security protocols, partial file transfer and management of parallelism for high speed transfers.

- *DUROC resource co-allocation library*

Most of the components of Globus toolkit work at quite low-level, and their main purpose is to serve as foundational blocks for higher-level middlewares and libraries, which are used for application development and deployment on Grids.

MPICH-G2 [KTF03] is Grid-enabled implementation of MPI [GLS99] message-passing communication library for distributed parallel applications. It is built on top of Globus toolkit, and it takes advantage of its GRAM and DUROC components. It includes routines both for point-to-point communication between two PEs (e.g. *MPI_Send* and *MPI_Receive* set of routines for synchronous, asynchronous, blocking, non-blocking or buffered sending and receiving of the data), and for collective operations over the set of PEs (e.g. *MPI_Broadcast*). MPICH-G2 enables parallel applications written using MPI standard to be executed on Grid infrastructure without any changes to their code.

2.1.2 Cloud Computing

Cloud computing is a relatively new branch of distributed computing that has gained a lot of attention recently.

There is no universally accepted definition of what the Cloud computing really is. For example, Vaquero et al. in [VRMCL09] compare over 20 different definition of Clouds. We will adopt the definition from their work, as it strives to find the minimum common denominator of characteristics of Clouds proposed by other definitions. Therefore, we will define Clouds as *large pool of easily usable and accessible virtualised resources (such as hardware, development platforms and/or services).* These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilisation. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customised Service-Level Agreements (SLAs).

USA National Institute of Science (NIST) definition of Clouds [NIS] lists essential characteristics that Cloud model possesses:

- *On-demand self-service.* A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider.

- *Broad network access.* Capabilities are available over the network and accessed through standard mechanisms, by a number of different client platforms (e.g. mobile phones, laptops and personal digital assistants (PDAs)).

- *Resource pooling.* The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand.

- *Rapid elasticity.* Capabilities can be rapidly and elastically provisioned, in some cases automatically, to quickly scale out and rapidly released to quickly scale in.

- *Measured Service.* Cloud systems automatically control and optimise resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g. storage, processing, bandwidth and active user accounts).

Just as there are many different definitions of Cloud computing, there are also many different surveys of the state-of-the-art in Cloud research, focusing on different aspects of Clouds (e.g. Zhang et al. [ZCB10], Rimal et al. [RCL09], Youseff et al. [YBDS08]). There is universal agreement in all of the surveys that there is very little standardisation in Cloud computing technologies. However, there are some common characteristics for all Cloud systems. Architecture of a cloud computing environment can roughly be divided into 4 layers (see Zhang et al. [ZCB10] (see Figure 2.1) :

- *The hardware layer* – the layer of physical resources of the cloud, including servers, routers, switches, power and cooling systems. Cloud physical resources mostly reside in large data centres.

- *The infrastructure layer* (or *virtualisation layer*) – this layer creates a pool of storage and computing resources by partitioning the physical resources using virtualisation techniques, such as VMWare [VMW] or Xen [Xen].

- *The platform layer*– where the infrastructure layer usually provides "raw" virtual machines, the platform layer build on top of it, adding operating system and application frameworks. The purpose of this layer is to provide users with higher-level frameworks for easier application development.

- *The application layer* – the layer of actual cloud applications.

Depending on which of these layers is offered as a service, cloud services are usually grouped into three categories [ZCB10]:

- *Infrastructure as a Service* – IaaS refers to on-demand provisioning of infrastructural resources, for example Amazon EC2 [Ama].

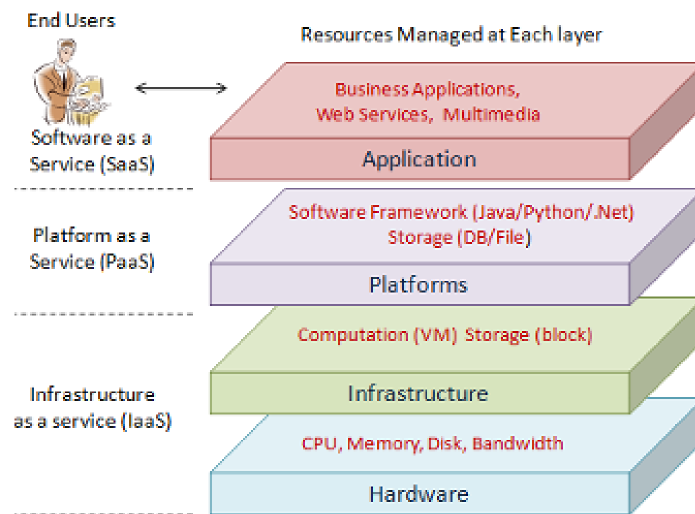


Figure 2.1: Cloud computing architecture (see Zhang et al. [ZBC10])

- *Platform as a Service* – PaaS refers to on-demand provision of platform layer resources, such as Google App Engine [Goo] framework for development of Web applications and Microsoft Windows Azure [MSA].
- *Software as a Service* – SaaS refers to on-demand provisioning of applications over the Internet, for example Salesforce.com [Sal].

Cloud Computing Products

In the following, we briefly describe some of the most prominent and successful cloud computing products:

- *Amazon Web Services* is a set of cloud services, which includes the following services:
  - *Amazon Elastic Compute Cloud (Amazon EC2)*, which allows user to create, execute and manage server instances, which are virtual machines running on top of XEN virtualisation engine. EC2 offers "bare" virtual machines, usually with just operating system installed, which offers users great flexibility in installing additional software and 'shaping' the way final machine is going to look like. EC2 also allows replicating virtual machines, by launching their identical copies.

  - *Amazon Simple Storage Service (Amazon S3)*, which is an online web storage service.
  - *Amazon CloudWatch*, which is a tool for collecting, processing and monitoring of raw data from other Amazon services, such as EC2.

- *Microsoft Windows Azure platform* consists of three components:
  - *Windows Azure* is a windows based environment for running applications and storing data on servers. It supports applications built on top of .NET framework, and also supports web applications built using ASP.NET and Windows Communication Foundation.
  - *SQL Azure* provides data services in the cloud based on SQL Server.
  - *.NET Services* supports the creation of distributed applications on the clouds.

- *Google App Engine* is a platform for web applications, which supports Python and Java. Deploying code, monitoring, managing fault tolerance and launching application instances is transparently handled by Google

Differences Between Cloud and Grid Computing

We can notice that there are a lot of similarities between Cloud and Grid computing. Indeed, Clouds and Grids share the same vision, which is that of reducing the cost of computation by aggregating physical resources in large, virtual supercomputers. It is not easy to define precisely what are the differences between the two technologies, mostly because there are no universally accepted definitions of either Grid or Cloud computing, and also due to a fact that there is very little standardisation in the Cloud computing field.

There is a lot of work (e.g. Foster et al. [FZRL08], Vaquero et al. [VRMCL09]) which tries to informally describe the differences between Grid and Cloud computing. Most authors agree that there is little technical difference, except for the fact that Clouds seem to rely much more on virtualisation of physical and logical resources than Girds do. Most of the differences, therefore, come from the different usage scenarios of Grids and Clouds. Foster et al. argue in [FZRL08] that currently Clouds are not as well suited for the execution of high-performance applications as Grids are. Grids and Clouds also have different security models, which is the consequence of the fact that resources in Clouds are centralised within a single organisation, whereas those in

Grids typically belong to different organisations. Clouds also tend to rely more on the economic model, where the usage is paid per unit of resource.

In order, however, to be able to define more precisely the differences between the two technologies, we will have to wait for standardisation of technologies and use-case scenarios to emerge in Clouds.

2.2 Scheduling on Distributed Computing Environments

Casavant and Kuhl in [CK88] define a general *scheduling problem* as a mechanism or policy used to efficiently and effectively manage the access to and use of a computing resource by its various consumers. Scheduling in general is concerned with the allocation of resources to multiple applications on the same computing environment, ensuring that certain Quality-of-Service (QoS) is achieved.

In our work, we are interested in the scheduling of tasks of a single parallel application on a distributed computing environment, that consists of a set of PEs connected by networks. Furthermore, we assume that all of the resources in the computing environment are fully dedicated to the execution of the application. This means that we will ignore some of the problems that can appear in, for example, scheduling on Computational Grids or Clouds, where resources are typically shared between multiple applications. Our focus is on the *load-balancing* decisions, which involve the decisions of how the work is migrated between different PEs in the system, so that the overall load balance is achieved. General scheduling algorithms, in addition to that, need to make decisions about what PEs will be allocated to what applications, initial placement of application tasks, a policy that will be used to choose what task will be executed on a PE which has multiple tasks available etc.

Following the taxonomy in Dong and Akl [DA06] and Casavant and Kuhl [CK88], scheduling algorithms can be classified by several criteria:

- *Local vs. Global* - Local scheduling is concerned with scheduling on a single resource. Global scheduling deals with multiple resources and allocates them to applications in order to achieve some global optimising goal.

- *Static vs. Dynamic* - In the static scheduling, all decisions are made at the start of application execution. In contrast to that, in dynamic scheduling the decisions are made 'on-the-fly', as the application execution progresses.

- *Optimal vs. Suboptimal* - Optimal algorithms make the decisions that are optimal with respect to some metrics (e.g. minimum application turnaround time, maximal resource utilisation). However, they require scheduler to have the perfect information about all resources and the applications it schedules. Thus, most of the research in scheduling is directed towards finding suboptimal solutions

- *Distributed vs. Centralised* - In the centralised scheduling, one centralised scheduler is responsible for making all the scheduling decisions. In distributed scheduling, the decisions are made by multiple distributed schedulers.

- *Cooperative vs. Non-Cooperative* - In the case of distributed scheduling, nodes that do scheduling can make cooperative or non-cooperative decisions. In the case of non-cooperative scheduling, individual schedulers are working autonomously and make decisions independent on their impact on the rest of the system. In cooperative scheduling, all schedulers are working towards a common system-wide goal.

According to this classification, load-balancing methods that we investigate in this thesis can be classified as *global, dynamic, distributed and suboptimal* scheduling. Policies described in Chapter 6 are *non-cooperative*, whereas the Feudal Work-stealing, presented in Chapter 5 is *cooperative.*

We will now present an overview of various scheduling methods for distributed computing environments, before focusing on load-balancing and, more specifically, work-stealing algorithms. Our main focus is on the Grid scheduling algorithms, since they typically assume a large-scale and highly heterogeneous computing environments.

Dong and Akl in [DA06] give a good and comprehensive overview of the Grid scheduling algorithms. In our discussion, we will consider two large groups of algorithms: those concerned with the scheduling of bag-of-tasks applications (i.e. applications that consist of totally independent tasks, all of which are available at the start of the application execution), and those concerned with the scheduling of applications with task dependencies.

2.2.1 Scheduling of Bag-of-tasks Applications

There is a lot of work that is concerned with the scheduling of the bag-of-tasks applications on the Grid. Similarity to our work lies in that most of these algorithms assume the irregularity in task sizes of an application, and they rely on the exact information about these sizes. The main difference between the bag-of-tasks applications and the

applications we consider in this thesis (see Section 4.2) is that the individual tasks in bag-of-tasks application are assumed to be sequential, whereas we also consider applications with nested parallelism. Another important difference is that all of the tasks of bag-of-tasks application are available at the beginning of the application execution, so the static scheduling of these tasks may be perfectly viable. In the applications we consider, on the other hand, only the main application task is available initially, and the additional tasks are created dynamically during the application execution.

Armstrong et al. [AHK98] propose several simple algorithms that map tasks of bag-of-tasks application to Grid resources, such as Opportunistic Load Balancing, which maps the first available task to the first available PE and Limited-Best Assignment, which assigns each task to the PE on which it has least-expected run time. Maheshwaran et al. [MAS+99] propose several more elaborate heuristics for the same problem:

- *Min-max* heuristic computes, for each task, its minimal execution time on all PEs. After that, the smallest task is placed on the PE which will execute it fastest, and the same procedure is repeated for the rest of the tasks.

- *Max-min* heuristic differs to Min-max in that, after the calculation of minimal execution time of each task, the *largest* task is placed on the PE that will execute it fastest.

- *Sufferage* heuristics computes what task would *suffer* the most (in terms of increased execution time) if it is not placed on the PE that will execute it fastest. After this task is found, it is placed on the PE that will execute it the fastest. The procedure is then repeated for the rest of the tasks

Casanova et al. [CLZB00] propose the XSufferage heuristics, which is the improvement of Sufferage that takes into account the fact that PEs are grouped into clusters. XSufferage calculates what task would suffer the most if it is not placed on its most preferable cluster (rather than the most preferable PE), and then places this task on the PE from its most preferable cluster that will execute it fastest.

All of the heuristics described above are relatively expensive in terms of computation time they require, and therefore they work well only for applications consisting of a relatively small number of coarse-grained tasks. In order to address the problem of applications comprising a large number of fine-grained tasks, Muthuvelu et al. [MLS+05] propose the heuristics that groups the fine-grained tasks into groups that are placed on the same PE. It takes into account total number of tasks, task sizes, total number of processors and the computing capabilities of processors in order to decide on the

grouping of tasks that will accomplish the minimal overall application execution time and maximal utilisation of Grid resources.

Besides static methods for scheduling such applications, there are also various methods that do dynamic scheduling, where not all of the tasks are assigned to all PEs initially. González-Vélez and Cole [GVC10] propose the statistical method for scheduling of divisible workloads, implemented in a task-farm algorithmic skeleton. Divisible workload is essentially a group of totally independent tasks, which conforms to the model of bag-of-tasks applications. The method proposed in their work, after an initial placement of a certain portion of application tasks to available PEs, dynamically, allocates the groups of tasks to PEs based on the performance of these PEs, and the variability of this performance.

A lot of research from mid 80s and the beginning of 90s dealt with the problem of scheduling the set of independent tasks to PEs, whereas assignment of tasks to PEs is done in packets. Packet can contain more than one task, and the main question was *how many* tasks to assign to a PE in one packet. We can see that this exactly corresponds to scheduling of bag-of-tasks applications. However, this research assumes the uniform communication latencies and computing capabilities of PEs in the computing environment. Static chunking (Hummel et al. [HSF92]) assigns all tasks at the beginning of the program execution. That is, if the initial number of tasks is n, and p is a number of PEs in a system, then n/p tasks are assigned to each. Self-Scheduling (Hummel et al. [HSF92]) assigns one task to each PE as soon as it gets idle (similar to work-stealing). Fixed-Size Chunking (Kruskal and Weiss [KW85]) always assigns tasks in chunks of equal size (where by 'size', it is meant the number of tasks in a chunk), and the optimal chunk size was determined to be $(\frac{\sqrt{2}nh}{\delta p\sqrt{\ln p}})^{2/3}$, where h is the overhead in transferring a packet to the destination PE, δ is the standard deviation of task sizes, n is the number of tasks and p is the number of PEs. A lot of other, more complicated and computationally more expensive algorithms have also been proposed (e.g. Bold algorithm in Hagerup [Hag97]).

2.2.2 Scheduling of Applications with Task Dependencies

The second large group of scheduling methods for heterogeneous distributed computing environments deals with the scheduling of applications where there are data or precedence dependencies between tasks. Such applications can be represented by directed acyclic graphs (DAG), which typically have weights assigned to their edges. Nodes of a DAG represent tasks, edges denotes precedence orders between tasks, and the weight of an edge denotes the amount of data that needs to be transferred between

preceding and succeeding tasks. Most of the DAG scheduling algorithms also rely on the accurate information about the sizes of individual tasks.

Compared to the applications we consider in this thesis, applications modelled by DAGs typically consist of a fewer number of coarse-grained tasks, and they involve a lot of data transfers between dependant tasks.

List scheduling heuristics assume that the tasks of a DAG are organised into priority list and iteratively assign tasks to resources, starting from the highest-priority tasks. Several list heuristics are compared in Grid setting in Zhang et al. [ZKK07]. Probably the most widely used one is Heterogeneous Earliest-Finish-Time (HEFT) heuristic, proposed in Tupcoglu et al. [THW02]. In HEFT, the priority of a task is defined as its distance from the exit node of the DAG, and the task with the highest priority is assigned to a PE that would compute it in the earliest time.

The most important drawback of HEFT is that it is computationally expensive. Radulescu and Germund in [RvG99] propose a Fast Critical Path heuristics, which avoids sorting all the task by priorities (as is done in HEFT), but instead maintains only a limited number of tasks sorted at any given time. Additionally, it does not consider all PEs as possible targets for a given task, but restricts the choice to either the PE from which the last message to the given task arrives or the PE that first becomes idle. This significantly reduces the complexity of HEFT heuristics, while maintaining good application execution times.

Sakellariou and Zhao in [SZ04] propose a hybrid heuristics for DAG scheduling, which partitions the input DAG into groups of independent tasks. These groups are then scheduled using Balanced Minimum Completion Time (BMCT) heuristics for scheduling of independent tasks. This hybrid heuristic is shown to perform very well, and it is less sensitive to the choice of methods for obtaining the weights of tasks in the DAG.

Darbha and Agrawal in [DA98] proposes a Task Duplication based Scheduling (TDS), which duplicates the execution of some tasks on idle processors in order to try to reduce the amount of communication needed for fetching the input data of some of their successor tasks.

To avoid large overheads that occur when large amount of data is transferred over high-latency networks, some DAG scheduling algorithms (for example, Dominant Sequence Clustering proposed in Yang et al. [YG94]) rely on clustering of heavily-communicating tasks, and allocating tasks from the same group to the same cluster of PEs (or, alternatively, to the same PE).

All of the algorithms described above assume a reliable setup, where there are no PE

failures or variations in the PE performance. Hernandez and Cole in [HC07b] propose Grid Task Positioning with Copying facilities (GTP/c) system, which starts from the initial schedule of DAG on a grid (obtained using list heuristics) and tries to improve it at fixed rescheduling points during the DAG execution. GTP/c allows migration of DAG tasks between Grid processors at rescheduling points. The decisions about task migrations are driven by the anticipated availability of grid processors, performance of communication links and the existence of multiple reusable copies of the same input data for tasks that may exist as a consequence of previous task migrations. GTP/c algorithm outperforms HEFT heuristics in the situations where grid resources vary considerably and data transfers between tasks are intensive. These authors extend GTP/c in [HC07a] with the *rewinding* mechanism, which recomputes and migrates tasks which disrupt the forward execution of succeeding tasks. This mechanism adds the fault tolerance to DAG scheduling, which is a very important feature in highly-dynamic Grids where failures of nodes are frequent.

Grid Workflow Systems

It is worth mentioning here that several systems tailored to Grids exist, which provide the whole environment for creation and execution of *workflows*. Workflows are essentially the applications with task dependencies, that can be modelled by DAGs. Some of these systems use DAG scheduling algorithms for mapping/remapping of tasks to available resources.

Yu and Buyya in [YB05] give a good taxonomy for Grid Workflow systems, together with the overview of current state-of-the-art systems.

GridFlow [CJSN03] is a workflow management system for Grids. It uses two-level hierarchical scheduling. A workflow is submitted to global grid for execution, where simulations are used to come up with the initial workflow schedule, where each sub-workflow is mapped to some local grid. This 'global' schedule can be changed due to a dynamic nature of the Grid environment (e.g. in the case of large delays of some sub-workflows, some parts of the workflow may be sent back to the simulator to compute the new schedule). After the initial schedule, each sub-workflow on local Grid is scheduled. This local scheduling takes into account granularity of tasks of a sub-workflow to resolve conflicts (i.e. if there are multiple tasks, possibly from different workflows, that need to executed at the same tame on the same resource). GridFlow uses the PACE toolkit [NKP+00] to predict the performance of a workflow task on some resource.

McCough et al. [MLD06] describe ICENI II (Imperial College e-Science Networked

Infrastructure), a service-oriented architecture for workflow component composition. In ICENI, an application is a composition of components and services defined using a workflow language. Users can specify their workflows with a high-level of abstraction, and ICENI II then makes the *realisation* of this workflow, and executes it on Grid resources. Scheduling of workflows is static, and matches the user requirements (in terms of, for example, time constraints on workflow makespan) with the available resources. ICENI II also supports checkpointing and migration of workflow components, as a response to, example, failure of Grid resources.

Grid Application Development Software (GrADS) project (Cooper et al. [BCC+05]) provides programming tools and an execution environment for the application development on Grids. An important part of GrADS project is a workflow scheduler, which resolves the application dependencies into a workflow and schedules the workflow components. Scheduling is done by choosing over several classic heuristics (Min-max, Max-min and Sufferage) the one that gives the minimum workflow makespan. GrADS also supports checkpointing and migration mechanisms, which are used to reschedule the workflow if a better schedule can be obtained at some point in its execution.

Pegasus (Planing for Execution on Grids, Deelman et al. [DBG+03]), is another workflow management and mapping system. It is used to map abstract workflows onto the available grid resources. This mapping is static. Jang et al. [JWT+04] propose a resource planner, which enables Pegasus to use the Prophesy (Taylor et al. [TWS03]) performance prediction mechanism to make better decisions about resources choices.

Application-Level Scheduling

An especially interesting class of scheduling methods for applications with interdependent tasks are *application-level* schedulers, which are tightly coupled to the specific application (or a class of applications). These scheduler can take advantage of the knowledge of application structure to make very efficient task schedules.

AppLeS (*App*lication *Le*vel *S*scheduling) project [BWC03] explored application-level scheduling for Grids. There, applications are extended with self-scheduling code for distributing and balancing the work across Grid resources. Based on that, schedule templates for parameter-sweep (APST - AppLeS Parameter Sweep Template [COBW00]) and master-worker (AMWAT - AppLeS Master-Worker Application Template [Sha01]) have been developed. Scheduling in APST consists of a scheduling of a set of independent tasks. APST scheduler sets the points in time at which the scheduling function is invoked. At each of this points, unallocated tasks are assigned to hosts for execution. Hosts on which tasks are executed are selected using either

self-scheduled work-queue algorithm, or some of the simple heuristics for list scheduling (Min-min, Max-min, Sufferage, XSufferage), whereas the scheduler dynamically changes the algorithm used. AMWAT scheduler lets the user choose which of the algorithms from predefined set (FIXED, Self-Scheduling, Fixed-size chunking, Guided self-scheduling, Trapezoidal self-scheduling, Factoring) to use for allocating tasks to nodes.

González-Vélez and Cole in [GVC06] describe the mechanism for scheduling the tasks of parallel pipeline application. The tasks represent the pipeline stages, and, after the initial mapping on PEs, they can be dynamically remapped as a response to the varying backload of PEs. The scheduler is incorporated into pipeline algorithmic skeleton.

2.3 Load Balancing

Especially interesting class of scheduling algorithms for us are load-balancing methods, which we outlined in Section 1.2. We have already classified these methods as global, dynamic, distributed and suboptimal scheduling methods. Load-balancing methods are especially well suited for distributed computing environments, because there is no centralised PE that does the scheduling (hence, there is inherently some level of fault-tolerance). Because of their principle of balancing the load between the PEs "on-the-fly", as the application execution progresses, these methods may be the only solution for scheduling of applications where dependencies between tasks are not known in advance, or where tasks are created dynamically during the application execution.

In Section 1.2, we have classified the load-balancing algorithms as either *sender-initiated* (*work-pushing*) or *receiver-initiated* (*work-stealing*). As a reminder, in sender-initiated algorithms a PE that has more than some predefined number of tasks in its task pool sends one (or more) tasks to an appropriately chosen PE. In receiver-initiated algorithms, an idle PE (a *thief*) asks the appropriately chosen PE (a victim) for work, and if the target has some tasks in its task pool, it sends one (or more) of them to the thief. Load-balancing algorithms usually differ between themselves in the way in which a PE that receives work (in work-pushing) or that is asked for work (in work-stealing) is chosen.

Good (if outdated) general overviews of load balancing techniques for homogeneous systems are given in Kumar et al. [KGV94] and Shivaratri et al. [SKS92].

In what is probably the most cited paper about work-pushing, Eager et al. [ELZ86] propose three different work-pushing algorithms:

- *Random*, where a sender sends one task to a random PE whenever it has some extra tasks)
- *Threshold*, where a sender polls a randomly selected PE, and if its task pool is smaller than some threshold, it sends one task to it
- *Shortest*, where a sender polls some set of PEs and sends a task to the one which has the shortest task queue.

Their conclusion was that Random algorithm works very well, and that the algorithms that use load information about PEs (Threshold and Shortest) cannot notably improve Random algorithm in homogeneous settings.

Shivartari et al. [SKS92] propose the combination between work-stealing and work-pushing, where a PE automatically offloads tasks if the number of tasks in its task pool reaches the upper-threshold value, and tries to steal work if the number of tasks falls below the lower-threshold value. This algorithm outperforms both Random Stealing and all work-pushing methods, but it relies heavily of frequent exchange of load information between PEs, in order to decide where to look for work and where to send work. This can be acceptable in low-latency computing environments, but is prohibitively expensive in high-latency ones.

There is not much work in evaluating the performance of work-pushing algorithms on heterogeneous computing environments. Van Nieuwpoort et al. [VNKB01] compare Random work-pushing with several work-stealing algorithm (including the Cluster-Aware Random Stealing), which we describe later. Their conclusion is that for divide-and-conquer applications, work-stealing gives much better speedups. We, therefore, in the rest of the thesis focus on work-stealing algorithms.

2.3.1 Work Stealing

Work stealing was first proposed in Burton and Sleep [BS81], in the context of the execution of functional programs on a virtual tree of processors. Nowadays, it is used in many runtime systems for the execution of parallel applications, both on shared-memory and on distributed computing environments.

All of the work-stealing algorithms we consider in this thesis use the same base algorithm, shown in Algorithm 1. The algorithm on the figure is executed independently by each PE in the computing environment. A PE first tests whether it has any tasks to execute. If it has, then it picks one of them and executes it. Alternatively, if no tasks are available, the PE sends the steal attempt to an appropriately chosen PE.

If the PE receives a steal attempt, and if it has more than one task, it chooses one (or more) of them and sends them to the thief that sent the steal attempt.

The simplest version of work-stealing algorithm is *Random Stealing* (Blumofe and Leiserson [BL99]), where targets for steal attempts are chosen at random, and only one task is sent as a response from victim. This algorithm works (provably) well for a class of strict computations and on very low-latency systems. This fact, together with the fact that it uses a distributed task pool, which reduces the amount of locking needed in shared-memory computing environments, deemed it suitable for using in runtime systems for shared-memory architectures (such as Cilk [BJK+95] and XWS [CKK+08] Work Stealing framework for X10 [Sar04] parallel programming language), and also for distributed computing environments with low latency between PEs (e.g. GUM [THM+96]). The main directions for the improvement of this algorithm on shared-memory computing environments are to further reduce or avoid locking operations on individual task queues, for example by allowing tasks to be stolen more than once [MVS09] and avoiding expensive task creation operations for each piece of work executed in parallel [HYUY09]. Also, another direction for improvement is controlling the data-locality, for example in [ABB00] with locality-guided work stealing algorithm.

Dinan et al. [DLS+09] recently showed that the Random Stealing algorithm scales very well on large (cca 8000 PEs) low-latency distributed computing environments, which also showed the practical applicability of Random Stealing on large-scale environments.

On the heterogeneous distributed computing environments, however, communication latencies between some of the PEs can be very high. Steal attempts can, therefore,

Algorithm 1 Basic work-stealing Algorithm

```
 1: repeat
 2:   if steal attempt received then
 3:     if have more than one task then
 4:       Choose one (or more tasks) and send them to the thief
 5:     else
 6:       Forward steal attempt to some other PE
 7:     end if
 8:   end if
 9:   if have a task to execute then
10:     Pick a task and execute it
11:   else
12:     Send steal attempt to some PE
13:   end if
14: until application execution finished
```

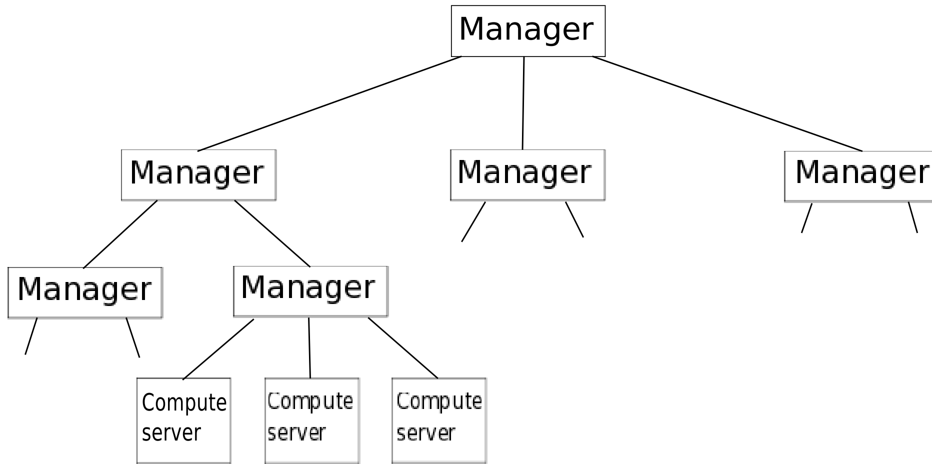


Figure 2.2: Hierarchical organisation of managers and compute servers in Atlas system

be very expensive, and it may not be tolerable for most of them to fail due to being sent to target that does not have any work to send back. Therefore, several algorithms have been developed that try to make better selection of steal targets on such environments. The most important ones, and the ones that we consider in this thesis, are various *hierarchical* work-stealing algorithms, Cluster-Aware Random Stealing, Adaptive Cluster-Aware Random Stealing and the Grid-GUM stealing. We describe the first three algorithms in detail below. The algorithm used in the Grid-GUM runtime environment is described in Section 2.4.3.

Hierarchical Work-stealing Algorithms

The Atlas (Baldeschwieler et al. [BBB96]) system architecture for global computing uses the *Hierarchical Stealing* algorithm. The system architecture consists of clients, managers and compute servers. A client submits the application to the local manager, which then finds an idle compute server where the application execution is started.

Managers and clients are organised into the tree (Figure 2.2 [BBB96]). Work-stealing in this hierarchical settings is done in the following way. When a compute server becomes idle, it performs random stealing in the cluster consisting of its siblings. Compute servers also periodically send their loads to their manager. Managers balance the load on the next level up. That is, manager waits until the whole its subtree is empty, and then steals work for its subtree from its manager and siblings. Similarly, when a subtree has too much work, it can allow stealing from above.

As pointed in Van Nieuwpoort et al. [VNKB01], this algorithm has two important drawbacks. The work stolen by manager nodes is stored on them rather on the

idle nodes, so additional steal messages are needed to distribute the work to its final destination. Also, the manager waits until the whole cluster is empty before looking for work from its siblings. This means that the whole cluster might stall while the manager fetches the work.

Javelin 3 (Neary and Capello [NC02]) is another system for developing large-scale parallel applications for heterogeneous environments. It uses another form of hierarchical work stealing. There, hosts that execute tasks are also organised in a tree. Each host, when it gets idle, first tries to steal tasks from its children, and, if that fails, it tries to steal from a parent node. This algorithm achieves good work locality, but, since it does not use the load information, large number of hosts need to be visited in order to obtain work. For example, if a host is looking for work into some subtree, than the whole subtree needs to be visited before search for work continues outside of it. When large parts of an environment are idle, this sends a lot of fruitless steal attempts. This algorithm can also suffer from stalling the whole subtree of hosts when its root is looking for work. This is the version of Hierarchical stealing that we will consider in comparing different work-stealing algorithms in Chapter 5.

Another hierarchical work stealing algorithm was proposed for divide-and-conquer applications developed using MPI-2 standard (Pezzi et al. [PCM+07]). Here, system is organised into a tree of manager and worker processes (similar to Atlas), where worker processes are leaves of the tree. Stealing is always attempted from child to parent in the tree. This mechanism suffers from the same drawbacks (slow work propagation from victim to thief via intermediate managers, possibly large number of managers visited to obtain work) as Javelin 3 work stealing.

(Adaptive) Cluster-Aware Random Stealing

Satin (Van Nieuwpoort et al. [VNWJB10]) is the system for developing divide-and-conquer parallel applications on Grid. It is based on Java, and it uses the *Cluster-Aware Random Stealing* (CRS) algorithm [VNKB01]. The idea of this algorithm is to steal the work from nodes in local and remote clusters *in parallel*. Whenever a PE becomes idle, it tries to steal some work from a node in a *remote* cluster. This is done asynchronously with the *local* stealing: instead of waiting for the result, the thief sets a flag and performs additional synchronous steal attempts within its own cluster. As long as the flag is set, only local stealing will be performed. When a reply from wide-area steal arrives, CRS resets the flag and, if the attempt was successful, adds the acquired work to its own task queue. This algorithm is compared to other state-of-the-art work stealing algorithms (including hierarchical ones) in [VNKB01],

and it was shown to perform better on computing environments consisting of multiple clusters where latency between different clusters is the same.

Further evaluation of this algorithm, in dynamic environments where latency and bandwidth vary over time, is presented in Van Nieuwpoort et al. [VNMW+04]. The conclusion there is that CRS algorithm performs less well on computing environments where communication latencies and bandwidth between different clusters are highly different (typical case in Computational Grids). Two improvements are proposed : Cluster-aware Multiple Random Stealing (CMRS), where nodes can send multiple wide-area steal requests to different clusters in parallel, and Adaptive Cluster-aware Random Stealing (ACRS), which attempts to actively detect and circumvent slow links. ACRS measures the time each wide-area steal attempt takes, and the chances of sending a steal attempt to a remote cluster depend on the performance of the WAN link to that cluster. CMRS performs even worse than CRS in scenarios with varying latencies and bandwidths between different clusters, whereas ACRS brings some improvements. We will consider both CRS and ACRS when comparing work-stealing algorithms in Chapter 5. Pseudocodes for these algorithms (i.e. functions that are executed when thief needs to choose steal target) are given in Algorithms 2 and 3 (where $lat(p, C)$ denotes the communication latency between pe p and cluster C).

Algorithm 2 Cluster-Aware Random Stealing

1: **if** not remoteStealing **then**
2: remoteStealing = **true**
3: Send steal attempt to a random remote target
4: **end if**
5: Send steal attempt to a random local target

Algorithm 3 Adaptive Cluster-Aware Random Stealing

1: **if** not remoteStealing **then**
2: remoteStealing = **true**
3: **for** each remote cluster C **do**
4: $P(C) = \frac{lat(thief,C)}{\sum_{D \in S} lat(thief,D)}$, where S is set of all remote clusters
5: **end for**
6: CL = random cluster from set of all cluster with probability vector P
7: p = random PE from cluster CL
8: Send steal attempt to p
9: **end if**
10: Send steal attempt to a random local target

2.4 (Parallel) Functional Programming

In functional programming languages, the principal building block is the function definition. The main program is a function which takes an input as an argument, and returns the output as a result, and is usually defined in terms of other functions, where high-order functions and functional composition are used as a "'glue"'. Furthermore, functions in functional programming languages are *pure*, in the sense that they do not allow side effects. They, therefore, conform to the mathematical definition of a function by transforming one input value into *exactly* one output value. This is known as a *referential transparency*, meaning that one expression can be replaced with the other one as long as they have the same value. This makes the functional languages especially suitable for mathematical reasoning and allows various optimisations for language compilers.

The most important advantage of functional programming is ease of modular design (Hughes [Hug90]). Two important features of functional languages that contribute to this are high-order functions and *lazy evaluation* (Field and Harisson [FH88]). Lazy evaluation is the implementation technique where an expression is evaluated only when it is actually needed, and to the level required. Thus, for example, if we have a function `f (x,y) = 2 + y` in a lazy language, then the argument `x` of a function would never be evaluated, since it is not mentioned in function body. Also, in a function `head` which returns the head of a generic list, only the first element of the list needs to be computed - the rest of the list is not important for this function. Lazy evaluation technique enables programmers to express programs that use potentially infinite data structures in easy and natural way.

Referential transparency renders functional languages very suitable for parallel programming. Hammond and Michaelson [HM99] note several reasons why is this true:

- **Ease of partitioning a parallel program**. In principle, any two subexpressions of functional program can be executed in parallel
- **Simple communication model**. Only communication channels that need to be introduced are result of data dependencies.
- **Straightforward semantic debugging**. Because semantic value of a functional program is independent of the evaluation order that is chosen, it doesn't change when it is executed in parallel.
- **Absence of deadlocks**. It is impossible to introduce deadlock by simply parallelising a functional program.

Consequently, in ideal case, writing a parallel functional program should differ from writing a sequential one 'only' in deciding what expressions *should* be evaluated in parallel (since every two of them *could* be evaluated in parallel). The programmer can be relieved of the most of the low level bookkeeping issues related to parallel threads (thread creation, communication, synchronisation, deadlock avoiding etc.). In reality, of course, the task of writing a *good* parallel functional program, which will deliver decent speedups on common parallel computing environments (such as multicore machines, clusters of workstations or Grids) is much harder than it looks. Expressions chosen for parallel evaluation need to be sufficiently coarse-grained and without too many data dependencies, or otherwise more will be lost in overheads related to the parallel evaluation (e.g. in creating threads, communication and data marshaling/unmarshaling) than is gained from parallel evaluation of expressions.

From the design point of view, parallel functional programming languages (and underlying runtime systems) can be classified, according to the level of control of parallel threads they give to the programmer into three categories (Loogen [Loo99]):

- **Implicit parallelism** approaches relieve all (or most) of the aspects of parallelism to the compiler and runtime system. They try to exploit parallelism that is inherent in the reduction semantics or the semantics of special operations (data parallelism). Programmer may only need to specify *what* expressions should be evaluated in parallel, or, in case of data parallel languages, to use special language constructs. Languages in this category range from fully implicit ones (for example, pH [NA01]), which require no programmer intervention whatsoever, to the ones which use special annotations for expressions that should be evaluated in parallel (e.g. Concurrent Clean [PvEPS99] and GpH [THLPJ98]). Data parallel languages (e.g. SISAL [Ske91], Data Parallel Haskell [CLJ$^+$07]) which provide special constructs for performing the same operation on many different data sets, also belong to this category.

- **Controlled or Semi-explicit** approaches expose some high-level constructs which can be used to express parallel algorithms. Parallelism is under control of programmer through these constructs, but low level details of their implementation are hidden. To this category belong, for example, *evaluation strategies* (Trinder et al. [THLPJ98]) built on top of GpH primitives for parallelism, GoldFish [Jay99] and NESL [Ble96] data parallel languages, and algorithmic skeletons (Cole [Col99]).

- **Explicit parallelism** approaches leave (almost) all of the control of the parallel

thread management (creating threads or processes, establishing communication channels between them etc.) to the programmer. In this way, they offer a more general programming model, compared to implicit and controlled approaches. Examples that belong to this category are Caliban [KT99], Eden [BLMP97], Scampi [Ś99] etc.

In this thesis, we will focus on implicit parallel functional language *Glasgow Parallel Haskell (GpH)*.

2.4.1 Glasgow Parallel Haskell

Glasgow Parallel Haskell (Trinder et al. [THLPJ98]) (GpH) is the modest extension to the non-strict lazy pure functional programming language Haskell (Peyton Jones and Hammond [Jon03]). It extends the Haskell language with two combinator related to the parallel execution: `par` and `seq`.

`par` combinator denotes the two expressions that *should* be evaluated in parallel. The expression `p ‘par‘ q` is semantically equivalent to `q`, and its dynamic effect is to indicate that it would be useful to evaluate `p` by a separate parallel thread, while the parent thread continues to evaluate `q`. If there is a free PE to 'grab' this expression, then the new thread will be created and `p` and `q` will be evaluated in parallel. However, if all PEs are busy, then the separate thread for `p` may not be created at all - expression `p` might be evaluated by a thread evaluating some other expression which refers to `p` or, in the case that `p` is not needed in the rest of the program, it might not be evaluated at all. Therefore, `par` should be seen only as an *indication* of usefulness of evaluating two expressions in parallel rather than a requirement.

`seq` represents the sequential composition operation. If the `p` is not $\perp$, the expression `p ‘seq‘ q` is semantically equivalent to the `q`, and its dynamic effect is to evaluate `p` to weak head-normal form (WHNF) before returning `q`. If `p` is $\perp$, then the value of whole expression is $\perp$.

2.4.2 GUM

GUM (Trinder et al. [THM+96]), which stands for "Graph reduction for a Unified Machine", is a portable parallel implementation of GpH designed for both shared and distributed memory systems. It is based on the GHC [JHH+92], the state-of-the-art sequential Haskell compiler which, in turn, uses the Spineless Tagless G-Machine (Peyton Jones [Jon92]) implementation of graph reduction. GUM extends the GHC runtime system to deal with `par` and `seq` constructs mentioned in Section 2.4.1.

Spark and Thread Management

The assumed underlying memory model in GUM is distributed-memory, where message passing is used for communication. Each PE has its own heap, and operates independently of all others. The basic unit of computation in GUM is a *thread*, purpose of which is evaluating the part of the graph. GUM thread is a logical object, and it is much more lightweight than the operating system thread. Thread is represented by a Thread State Object (TSO) structure. TSO contains the information about thread's current state (whether it is runnable or blocked, its registers etc.), together with the thread's stack. Each PE holds its own pools of runnable and blocked threads. In addition to that, each PE also has its own *spark pool*. *Spark* is a pointer to the part of the graph denoted for parallel evaluation using the `par` construct. Thus, whenever a thread encounters `p ‘par‘ q` expression in its evaluation, a spark for `p` is placed in a spark pool of PE evaluating it, and the thread continues to evaluate `q`. Whenever a pool of runnable threads of a PE becomes empty, it chooses a spark from its spark pool (in a FCFS manner), turns it into a thread and starts evaluating it.

GUM uses the Random Stealing load-balancing algorithm. At the beginning of the application execution, one PE is nominated as a *main* PE, and it starts evaluating the main thread (which, in turn, evaluates the application's main function). All other PEs are idle. Whenever a PE is idle (i.e. it becomes a thief), it starts looking for work by sending a `FISH` message to a randomly chosen target PE. If the target PE has some sparks in its spark pool, it chooses one of them (again in FCFS manner) and offloads it (together with some 'nearby' graph) to the thief in a `SCHEDULE` message. Upon receiving the `SCHEDULE` message, the thief unpacks it and adds the spark from it to its spark pool. If the target PE does not have any sparks in its spark pool, it forwards the `FISH` message to some other (again, randomly chosen) PE. The number of PEs that a single `FISH` message can visit is bounded by the `maxAge` parameter. After `maxAge` PEs are visited, and if no work is found in any of them, `FISH` message is returned to the thief that initiated it.

Distributing Data

Since Haskell is a lazy programming language, GUM needs to take care that each expression is evaluated *at most* once, even in parallel settings. It is possible for multiple sparks to share the same subexpression, and if two (or more) of them are turned into threads, care must be taken that two threads do not end up evaluating this subexpression. This is done using two mechanisms - *blackholing* (if conflicting threads reside on the same PE) and *lazy data fetching* (if they reside on different PEs):

- **Blackholing:** When a thread starts evaluating a thunk (which represents the suspended computation), the thunk is overwritten by a *black hole*. If some other thread encounters this black hole during its evaluation, it saves its state in its TSO, attaches it to the black hole's *blocking queue* and blocks. Once the evaluation of the original thunk is finished, the black hole is overwritten with a normal form value, and all TSOs from its blocking queue are moved to the pool of runnable threads.

- **Lazy data fetching:** In the case that one of the two sparks that share the same subexpression is offloaded to a remote PE (and turned into a thread there), *lazy data fetching* comes into the game. When a victim packs some subgraph of a graph into a packet, in order to send it to the thief, care is taken that exactly one copy of each thunk that is encountered in packing exists, either in the victim's heap, or in the packet. The other copy is substituted with a `FETCH_ME` closure. `FETCH_ME` closures are global pointers to the data, and they contain a *global address* that uniquely identifies the PE and the position in its heap where the data resides.

  The `FETCH_ME` closures are synchronisation points for the threads residing on different PEs. When a thread, during its evaluation, encounters the `FETCH_ME` closure, it sends the request for the corresponding data to a PE where this data resides. This is done via `FETCH` message. Subsequently, the thread blocks. When the PE holding the data receives the `FETCH` message, two situations can happen:

  – The data may already be evaluated to normal form, in which case it is packed (again, together with some nearby graph) and sent back to the PE requesting it in a `REPLY` message.

  – Alternatively, the data might be under evaluation (blackholed), in which case the fetch request is suspended (`BLOCKED_FETCH` is created) and added to the black hole's blocking queue. Once the blackhole is updated with a normal form value, this fetch request is unblocked and processed.

The questions of the size of a packet (in `RESUME` and `SCHEDULE` messages), and how many thunks to put into it are investigated in Loidl and Hammond [LH97].

To summarise, each PE in GUM executes the main scheduling loop given in Algorithm 4 until either the main thread finishes or some error is encountered. The overall GUM load distribution scheme is given in Figure 2.3.

Figure 2.3: GUM Load Distribution Scheme [THM+96]

Improvements to the basic GUM load-balancing mechanism

There have been various attempts to improve the basic load distribution mechanism used in GUM. Loidl in [Loi02] considers thread and spark pool *watermarks*, which represent the lower and upper limit on the number of threads/sparks that can be in

Algorithm 4 schedulingLoopStep() function

1: **repeat**
2: processMessages()
3: **if** run queue empty **then**
4: **if** have sparks in spark pool **then**
5: Choose spark s for local execution
6: Turn s into a thread t
7: Add t to run queue
8: **else**
9: d = choose a random PE
10: Send FISH message to PE d
11: **end if**
12: **end if**
13: **if** run queue not empty **then**
14: Run one of the threads in run queue
15: **if** thread blocked on remote data **then**
16: Send FETCH message
17: **end if**
18: **end if**
19: **until** main thread finishes or error encountered

corresponding pools. Whenever a number of threads in all of the PE's queues is lower than the low-watermark, the PE will try to create additional thread, even if it has some runnable ones. Conversely, if the number of threads in thread queues is higher than the high-watermark, the PE will not try to an additional ones, even if it has no other thread to evaluate. Similarly, if the number of sparks in the PE's spark pool is less then spark low-watermark, the PE will try to steal new sparks. These mechanisms are particularly usable in the applications where there is a danger of one PE (usually the main one) grabbing all the sparks and turning them into threads. This can happen if sparks share some portion of a graph which is under evaluation, so each time a new spark is turned into a thread, this thread will block immediately, forcing a PE to create additional threads. High-watermark for threads can prevent this situation.

In the basic GUM system, only sparks can be migrated between PEs. Du Bois et al. in [BLT03] describe the implementation of *thread* migration mechanism in GUM, which allows the migration of already started threads between PEs. This is shown to be crucial in some situations similar to the one we described when discussing spark and thread watermarks.

In [Loi98], Loidl investigates the usage of granularity information in making some of the scheduling decisions in GUM. Specifically, he considers *priority sparking*, where a spark for thunk is created only if its granularity is large enough, and the *priority scheduling*, where the largest thread from the PE's runnable pool is selected for evaluation in each step of main scheduling loop. He shows that these mechanisms strategies can bring improvements for some irregular parallel applications on homogeneous systems.

2.4.3 Grid-GUM

Grid-GUM [AZ06] is the extension to the GUM runtime system. Grid-GUM aims to adapt GUM to the Grids. It is build on top of Globus [Fos03] Grid middleware and it uses the MPICH-G2 [KTF03] implementation of MPI communication library.

The main difference between Grid-GUM and GUM lies in the work-stealing algorithm used for load balancing. Grid-GUM replaces the Random Stealing algorithm used in GUM (which performs very well on low-latency homogeneous systems, but badly on Computational Grids) with the advanced, adaptive work-stealing mechanism, based on the cheap exchange of load information between PEs and its use in the selection of steal targets. It accounts for different latencies between the PEs and different PE computing capabilities, and tries to adapt the load distribution to it. The load-balancing algorithm is described in detail in Al Zain et al. [AZTML06] and eval-

uated in Al Zain et al. [AZTML08]. Since this algorithm is important for this thesis, as it uses the load information in making steal decisions, we now describe it in more detail.

Grid-GUM assumes that the set of all PEs is organised into a set of clusters. Compared to the GUM work stealing, Grid-GUM makes the following improvements :

- The fastest PE (in terms of computing capability) from the largest cluster is nominated as the main PE. Since the assumption is that the main thread will generate the most of the parallelism (typical situation in divide-and-conquer and data-parallel applications), in this way it is ensured that the most work is generated in the largest cluster.

- Grid-GUM differs between steal attempts (`FISH` messages) that arrive from the same cluster and the ones from remote clusters. If the `FISH` arrives from the same cluster, one spark is sent as a response in a `SCHEDULE` message (as is done in GUM). On the other hand, if it originates from a different cluster, then Grid-GUM tries to send more sparks in the single `SCHEDULE` message, to cover the possibly high latency. In this case, it tries to balance the number of sparks (relative to the PE computing capability) between the two PEs involved in the steal operation.

- In Grid-GUM, the PEs exchange information about the load of the environment as the application execution progresses. The load information (the number of sparks and threads that each PE has in its pools, together with the timestamps when this information was last updated) that a PE has is attached to each `FISH` and `SCHEDULE` message sent by it. When some other PE receives the `FISH` or `SCHEDULE` message, it compares the timestamps of the load information from the message with the timestamps of its own load information, and updates the latter one for each PE for which it received a more recent information. Conversely, the load information in the message is also updated for all PEs for which the PE that receives it posses more recent information. In this way, hopefully, every PE in the environment will have the accurate information about the load of every other PE. When a PE gets idle, it will know where it is most likely to find some work, and it chooses the target for `FISH` message accordingly (see below). It is worth noting that, since the load information is attached to `FISH` and `SCHEDULE` messages that would be exchanged anyway, the exchange of load information in Grid-GUM comes at almost no extra cost (except for the small constant additional overhead in processing these two types of messages).

- Each PE also has a communication map table, which holds the current communication latencies between that PE and every other PE in the environment (again, together with the timestamps of these information). This table is consulted when the PE needs to choose the destination for sending (or forwarding) a `FISH` message. If the PE 'knows' about more than one other PE that has some work to offload, it will prefer sending a `FISH` message to the one which is *the nearest* to it (i.e. the one for which the communication latency to it is the lowest).

The Algorithms 5 and 6 (taken from Al Zain [AZ06]) give the pseudocodes for the Grid-GUM scheduling loop and the function that processes the `FISH` message (the central parts of Grid-GUM adaptive load balancing system).

Algorithm 5 Grid-GUM schedulingLoopStep() function

```
1: repeat
2:   processMessages()
3:   if run queue empty then
4:     if no sparks in spark pool > low-watermark then
5:       Choose spark s for local execution
6:       Turn s into a thread t
7:       Add t to run queue
8:     else
9:       update local load info
10:      calculate localRatio=localSpeed/localLoad
11:      sort communication latencies om ascending order
12:      for each destPE in a system do
13:        calculate destRatio = destSpeed/destLoad
14:        if localRatio>destRatio then
15:          attach to FISH message my system load information (together with
             timestamps)
16:        end if
17:        send FISH message to destPE
18:        break
19:      end for
20:      if not main PE and no FISH message sent then
21:        destPE = mainPE
22:        attach to FISH message my system load information (together with times-
           tamps)
23:        Send FISH message to destPE
24:      end if
25:    end if
26:  end if
27:  if run queue not empty then
28:    Run one of the threads in run queue
29:    if thread blocked on remote data then
30:      Send FETCH message
31:    end if
32:  end if
33: until main thread finishes or error encountered
```

Algorithm 6 Grid-GUM processFISHMessage() function

```
1: update communication map from FISH message
2: update system load information from FISH message
3: if spark pool non-empty then
4:   calculate localRatio (=localLoad/localSpeed) and originRatio (=originLoad/originSpeed)
5:   if originRatio > localRatio then
6:     if originPE and localPE withing the same cluster then
7:       send one spark in SCHEDULE message to originPE
8:     else
9:       localClusterRatio=localClusterLoad/localClusterSpeed
10:      originClusterRatio=originClusterLoad/originClusterSpeed
11:      calculate noSparksToSend
12:      send noSparksToSend sparks to originPE
13:    end if
14:  else
15:    if FISH exceeded age then
16:      return FISH to originPE
17:    else
18:      for each destPE in a system do
19:        calculate destRatio
20:        if originRatio>destRatio then
21:          send FISH (together with system load information and timestamps) to destPE
22:          break
23:        end if
24:      end for
25:      if not mainPE and no FISH message sent then
26:        send FISH (together with system load information and timestamps) to mainPE
27:      end if
28:    end if
29:  end if
30: end if
```

Chapter 3

SCALES Work-Stealing Simulator

In this chapter, we present SCALES, a work-stealing simulator for parallel applications on distributed computing environments. Section 3.1 gives an overview of SCALES. Sections 3.2 and 3.3 describe, respectively, the applications and the computing environments that SCALES can simulate, and Section 3.4 describes how the application execution is simulated. Section 3.5 gives an overview of other simulators for heterogeneous computing environments.

3.1 Overview of SCALES

SCALES can simulate the execution of a wide class of parallel applications using different work-stealing algorithms. Currently, it supports Random, Hierarchical, Cluster-Aware Random, Adaptive Cluster-Aware Random, Grid-GUM and Feudal (see Section 5.3 for more details) work-stealing algorithms. To implement a new work-stealing algorithm in SCALES, we simply need to implement the functions that deal with the selection of steal targets.

The level of abstraction provided by SCALES enables us to model applications which conform to many parallel programming paradigms, including divide-and-conquer, master-worker and data-parallel applications. One important restriction, however, is that the application modelled cannot have any non-trivial data-dependencies between the tasks forked by the same parent task. It is also possible to simulate heterogeneous computing environments, where heterogeneity comes from different communication latencies between different PE pairs as well as different computing capabilities of PEs. Additionally, SCALES allows the tuning of overhead which certain operations take during the application execution (such as processing messages, task creation and task finish). This enables users to fine-tune the simulator, in order to model more accurately

specific runtime systems.

The following figure shows the output of an example run of SCALES simulator:

```
> sim_crs -p SimpleDC_10_5ms -s Grid_8_8
Total time 386002
Successful/Attempted steal messages ratio 0.048302
--
Tasks executed in cluster 0 - 687
Tasks executed in cluster 1 - 506
Tasks executed in cluster 2 - 567
Tasks executed in cluster 3 - 425
Tasks executed in cluster 4 - 453
Tasks executed in cluster 5 - 465
Tasks executed in cluster 6 - 491
Tasks executed in cluster 7 - 500
```

For the simulation of each application execution, SCALES requires three pieces of information:

- work-stealing algorithm being simulated (`sim_crs` in the above example, which represents Cluster-Aware Random stealing)
- the file with the application model (`SimpleDC_10_5ms` in the above example, see Section 4.2 for details)
- the file with the computing-environment model (`Grid_8_8` in the above example, see Section 5.2).

SCALES returns the number of time units that the application execution takes. It also returns some statistics about the steal messages sent and the tasks executed. Precisely which statistics are to be returned is decided by user. In the above example, we have chosen to display the ratio between all steal messages sent and those successful in obtaining work. We have also chosen to display the number of tasks executed in each cluster, in order to see how evenly the load was spread across the environment.

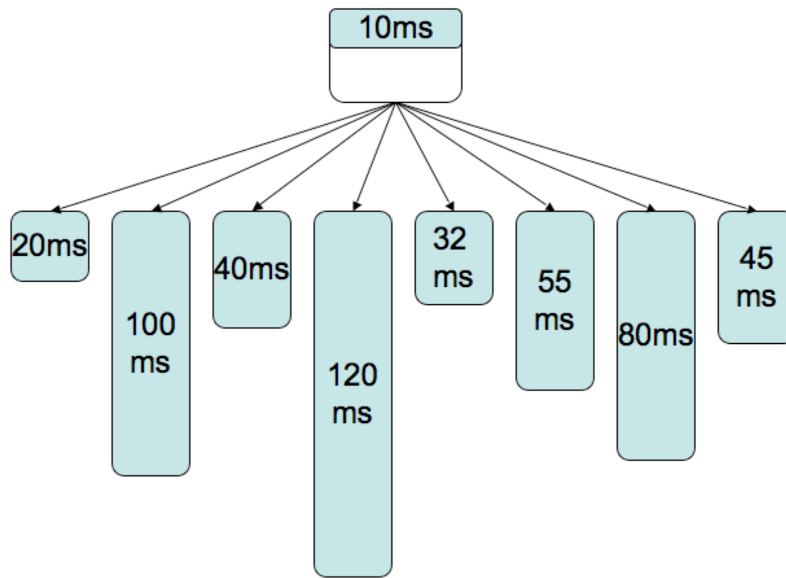


Figure 3.1: Task graph of a simple data-parallel application

3.2 Applications

In the SCALES application model, each application task consists of a list of *task events*. There are two types of task events:

- RUN r event, where the task runs for r *relative* time units.
- FORK $(t_1, \ldots, t_n)$ event, where the task forks a set of child tasks $\{t_1, \ldots, t_n\}$ and then blocks until all of them finish execution.

An application in SCALES consists of a *main* task, which is the only task available at the beginning of application execution.

With this level of abstraction, we can model applications that conform to many popular parallel programming paradigms. For example, consider a simple data-parallel application on Figure 3.1. In this application, the main task first does the initial computation, which takes 10ms. Subsequently, it creates 8 subtasks and then blocks. After all of the subtasks finish execution, the main task also finishes. The sizes of subtasks are 20ms, 100ms, 40ms, 120ms, 32ms, 55ms, 80ms and 45ms, respectively. We can model this application with the following file:

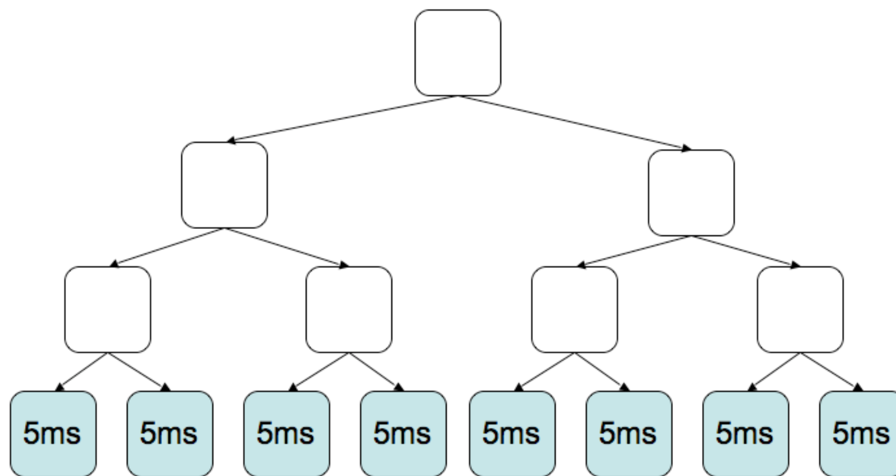


Figure 3.2: Task graph of a simple divide-and-conquer

```
{
 RUN 10000,
 FORK
     {RUN 20000}
     {RUN 100000}
     {RUN 40000}
     {RUN 120000}
     {RUN 32000}
     {RUN 55000}
     {RUN 80000}
     {RUN 45000}
}
```

Note that the time units in the above example are microseconds. The curly brackets surround the individual tasks, and the commas separate the task events.

As another example, Figure 3.2 shows an example of a simple divide-and-conquer application. In this application, the "divide" and "conquer" parts are trivial (i.e. they take a negligible amount of time) and all tasks generated after the third level of recursion is reached are sequential. Each sequential task takes 5ms to complete. This application can be modelled by the following file:

```
{
  RUN 1,
  FORK {RUN 1,
          FORK {RUN 1,
                  FORK {RUN 5000}
                         {RUN 5000},
                  RUN 1
                 }
                 {
                  RUN 1,
                  FORK {RUN 5000}
                         {RUN 5000},
                  RUN 1
                 },
          RUN 1
         }
         {RUN 1,
          FORK {RUN 1,
                  FORK {RUN 5000}
                         {RUN 5000},
                  RUN 1
                 }
                 {RUN 1,
                  FORK {RUN 5000}
                         {RUN 5000},
                  RUN 1
                 },
          RUN 1
         },
  RUN 1
}
```

Similarly to the examples shown above, we can model divide-and-conquer applications with non-trivial “divide” or “conquer” phases, data-parallel applications with nested parallelism, master-worker applications etc. However, one thing that we can observe from the SCALES application model is that there is no task event that simulates the communication between tasks. The only communication that takes place is between

parent and child tasks, and this communication is implicit in the model. A child task fetches the data from the parent task before its execution starts, and it sends the result back after its execution ends. This restricts the applications that can be modelled under SCALES to those that are *embarrassingly parallel*.

3.3 Computing Environments

In the SCALES model, a computing environment consists of a set of PEs that can execute tasks. PEs are grouped into *clusters*, and the clusters are connected by networks. The communication latency between each two PEs from the same cluster is the same, and they all have the same computing capability. In the rest of the text, we will refer to communication latency simply as *latency*, since this is the only kind of latency that we consider.

To describe the computing environment being simulated, we need to provide the description of each cluster in it, together with the latencies between these clusters. For each cluster, we need to provide two parameters: the number of PEs in it, and the computing capability of its PEs. The computing capability of a PE is specified as a real number, which denotes how many relative time units of RUN task event it executes in one *absolute* time unit. Therefore, a PE with the computing capability of 2 will execute RUN 4000 event in 2000 absolute time units, whereas a PE with the computing capability of 0.5 will execute the same event in 8000 absolute time units. The latency between two clusters denotes the number of absolute time units it takes to send a message from one cluster to the other. It needs to be specified for all pairs of clusters. The latency inside a cluster (i.e. between two PEs from it) is specified as the latency between that cluster and itself.

Consider, for example, a computing environment that consists of 8 clusters, where each cluster is denoted by an integer. Every cluster consists of 8 PEs. The latency inside each cluster is 0.1ms, and the clusters are split into two continental groups of clusters – clusters 0..3 belong to the first continental group, whereas clusters 4..7 belong to the second. The latency between every two clusters that belong to the different continental groups is 80ms. Furthermore, each PE from the first continental group is twice as fast as any PE from the other continental group. Each continental group is further divided into two country groups (so clusters 0 and 1 belong to the first country group, and clusters 2 and 3 to the second, and similarly for clusters 4, 5, 6 and 7). The latency between every two clusters from the same continental group, but different country groups, is 30ms, and the latency between every two clusters from

the same country group is 10ms. This computing environment can be described by the following file (with added comments):

```
8                             # number of clusters
8 8 8 8 8 8 8 8               # number of PEs in each cluster
1 1 1 1 0.5 0.5 0.5 0.5       # computing capabilities of the clusters
0 0 100                       # latency inside cluster 0
1 0 10000                     # latency between clusters 1 and 0
1 1 100                       # latency inside cluster 1
2 0 30000                     # latency between clusters 2 and 0
2 1 30000                     # ...
2 2 100
3 0 30000
3 1 30000
3 2 10000
3 3 100
4 0 80000
4 1 80000
4 2 80000
4 3 80000
4 4 100
5 0 80000
5 1 80000
5 2 80000
5 3 80000
5 4 10000
6 0 80000
6 1 80000
6 2 80000
6 3 80000
6 4 30000
6 5 30000
6 6 100
7 0 80000
7 1 80000
7 2 80000
7 3 80000
```

7 4 30000
7 5 30000
7 6 10000

We can see that SCALES supports modelling of the environments that are heterogeneous in terms of the latencies between PEs and/or the computing capabilities of individual PEs. However, it does not support modelling of dynamically changing latencies between the same clusters, or the dynamic variability in the computing capability of the same PE. Additionally, it also assumes the uniform bandwidth for all networks. Therefore, it cannot fully model highly dynamic computing environments, such as Computational Grids. The reason for making these assumption is that we are interested in the scenario where the whole computing environment is fully dedicated to the execution of a single application. In this scenario, the probability of an unexpected congestion of networks or of variable performance of individual PEs (due to a, say, increased backload) is low.

3.4 Execution of Applications under SCALES

In SCALES, each PE has three *task queues* associated with it (terminology for these queues is adopted from the GUM runtime environment):

- *Spark task queue*, denoting the tasks whose execution still has not started (*unstarted* tasks).

- *Run task queue*, denoting the tasks whose execution has already started (*started* tasks) and that are ready to continue execution.

- *Blocked task queue*, denoting the started tasks that are blocked on their subtasks.

The execution of an application under SCALES starts with the main task in the run task queue of some random PE, and all task queues of other PEs are empty. Each PE then independently runs the scheduling loop (described in Algorithm 7).

We will now analyse in more detail how the particular steps of the scheduling loop are carried out. Note that the execution of a step may be interrupted if a new message arrives to a PE, in which case the code for processing messages (described in Algorithm 8) is executed, and after that control returns to the step which was interrupted.

In step 3, a PE chooses a task from its task queue that will be executed next[1]. We assume that the task is chosen in Last-Come-First-Served manner, i.e. the task that is last added to the run queue is the first one to be executed. This follows the general principle of work-stealing algorithms. The same holds for step 19, where the PE needs to choose a task to transfer from its spark queue to its task queue. However, in Chapter 6, we consider other task selection policies, where the tasks at this step are selected according to their size, rather than their age.

In step 4, a PE needs to run the next task event e. The way in which this is carried out depends on the type of e:

- If e is a RUN r event, then the PE simply runs for r/c absolute time units, where c is its computing capability. However, the execution of a RUN event can

[1]Note that, since a PE never moves tasks from the spark task queue to the run task queue if the run task queue is nonempty, the only situation where more than one task exists in the run task queue is if tasks are moved to it from the blocked task queue.

Algorithm 7 SCALES Scheduling Loop

```
 1: while main task not finished do
 2:   if run task queue non-empty then
 3:     Choose a task t from the run task queue
 4:     Execute the first event e from the list of events of task t
 5:     if the execution of e was not preempted then
 6:       Remove e from the list of events t
 7:     end if
 8:     if list of events of t is empty then
 9:       Post-process the task t
10:       Remove task t
11:     else
12:       if e was fork event then
13:         Move t to the blocked task queue
14:       else if e was run event then
15:         Move t to the run task queue
16:       end if
17:     end if
18:   else if spark task queue non-empty then
19:     Choose a task t from the spark task queue
20:     Move t to the run task queue
21:   else
22:     if not looking for work then
23:       Look for work
24:     end if
25:   end if
26: end while
```

be interrupted at any point, by the arrival of a message. In this case, if the PE has been executing the RUN r event for s absolute time units before it was interrupted, then the event RUN $(r - c \cdot s)$ replaces the event RUN r at the beginning of the list of events of the task t.

- If e is the FORK $(t_1, \ldots, t_n)$ event, then the tasks $t_1 \ldots t_n$ are placed into the spark task queue of the PE. The execution of a FORK event takes 0 absolute time units, and, therefore, it cannot be interrupted. After the execution of the FORK event, the task t that is being executed is placed in the blocked task queue (step 13). t is unblocked (i.e. moved back to the run task queue) when all of the tasks $t_1, \ldots, t_n$ finish their execution and send their results back. This implies that we need to track down which of these tasks have finished execution. To do this, when t is moved to the blocked task queue, we attach to it a list of its unfinished subtasks. This list initially consists of tasks $t_1 \ldots t_n$, and it is updated either in step 10 or when a *reply* message arrives (see below).

In step 9, a PE (we will denote it by T) needs to post-process the task t that has finished its execution. If the task t is the main task, then the whole application execution is finished, and the simulation ends. Otherwise, t was forked by its parent task t_0. Let T_0 denote the PE that has t_0 in its blocked task queue. If T and T_0 are the same PE, then t is removed from the list of unfinished subtasks of t_0. If this results in the list of unfinished subtasks of t_0 becoming empty, then t_0 is moved to the run task queue of PE T. In the case that T and T_0 are not the same PE, then a reply message that contains task t is sent from T to T_0.

Algorithm 8 shows the function for processing messages. The way in which individual messages are processed depends on their types. In SCALES, there are 3 different types of messages that a PE can receive:

- A *fish* message is of the form FISH o, and it is the message by which thief o asks the target for work. The receiver responds to this message by choosing n tasks (where n depends on the work-stealing algorithm simulated) from its spark task queue and sending them in a *schedule* message to the thief o. In the case that the receiver's spark task queue is empty, the receiver forwards the message to some other appropriately chosen PE (again, the way in which this PE is chosen depends on the work-stealing algorithm simulated). Alternatively, if the fish message has already visited `MAX_PES` PEs, the receiver returns it to the thief o. `MAX_PES` is a parameter which depends on the work-stealing algorithm used. In most cases, we set this parameter to be equal to the number of PEs in the

environment.

- A *schedule* message is of the form SCHEDULE $(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are tasks. This is the message sent from a victim to a thief in the case that steal attempt is successful. The receiver of the schedule message processes it by placing tasks $t_1 \ldots t_n$ into its spark task queue.

- A *reply* message is of the form REPLY t, where t is a task from the list of unfinished tasks of some task t_0. t_0 resides in the receiver's blocked task queue. This message is processed by removing task t from the list of unfinished tasks of task t_0 and, if this list is now empty, placing the task t_0 into the receiver's run task queue.

From the description of the SCALES scheduling loop, we can observe that different work-stealing algorithms have different implementations of steps 22 and 23 of the scheduling loop, and different implementations of the function that processes messages. Specifically, they differ in where they forward the fish when processing FISH message (step 7 of the processMessage function), how they decide whether a PE is stealing or not in step 22 and where they look for work in step 23 of the scheduling loop. In Random Stealing, for example, the fish message is forwarded to a random PE; in step

Algorithm 8 processMessage(PE pe)

1: **if** message is FISH o **then**
2:    **if** pe's spark task queue empty **then**
3:       $\{t_1, \ldots, t_n\}$=Choose tasks from pe's spark task queue
4:       Send SCHEDULE $\{t_1, \ldots, t_n\}$ message to o
5:    **else**
6:       **if** FISH message visited MAX_PE PEs **then**
7:          Send FISH message to an appropriately chosen PE
8:       **else**
9:          Send FISH to o
10:       **end if**
11:    **end if**
12: **else if** message is SCHEDULE $\{t_1, \ldots, t_m\}$ **then**
13:    Place tasks $t_1, \ldots, t_m$ in pe's spark task queue
14: **else if** message is REPLY t **then**
15:    t_0 = parent task of t
16:    Remove t from t_o's unfinished tasks queue
17:    **if** t_o's unfinished tasks queue empty **then**
18:       Move t_0 to pe's run task queue
19:    **end if**
20: **end if**

23, a thief looks for work by sending a fish message to a random PE; and step 22 consists of testing whether a PE has any outstanding fish messages. In Cluster-aware Random Stealing, on the other hand, the fish message is forwarded to a PE from the same cluster. In step 23, a thief looks for work by sending a *local* fish message (i.e. fish message sent to a target from the same cluster), as well a *remote* fish message (i.e. fish message sent to a target from a different cluster). Remote stealing is done only if no other outstanding remote fish message from the same thief exist. Step 22 consists of testing whether a thief has an outstanding local fish message.

We can observe several characteristics of SCALES mechanism for simulating the application execution. Firstly, by default the scheduling is preemptive, which means that the execution of a task is immediately suspended when a PE receives a new message. User can supply a command line switch, which makes the scheduling non-preemptive. The decision of whether to use preemptive or non-preemptive scheduling depends on the particular runtime system that we want to simulate. For example, Grid-GUM runtime system is by default non-preemptive. A thread will stop its evaluation only in the cases when either garbage collection is needed, or the evaluation has finished. However, garbage collections occur relatively frequently, so threads will often need to stop evaluation. In this way, messages will usually be processed shortly after they are received. Therefore, we have found out that preemptive scheduling in SCALES better simulates the execution of applications under Grid-GUM.

The second characteristic of SCALES that we can observe is that if a target receives a fish message and if it has no work to offload, it will, in most cases, forward that message to some other target. The alternative approach, which some runtime systems use, is to send the fish message back to the thief. We have found out that the former approach results in much better load balance, since many fewer stealing-related messages need to be sent.

3.4.1 Accuracy of Simulations Under SCALES

In order to test the accuracy of simulations under SCALES, we have implemented the tracing mechanism, which can be turned on by the `-r` command-line switch. With this switch, the simulation of the application execution under SCALES produces an extensive trace file, where all actions done by all of the PEs in the environment are logged. This file can then be inspected (or some tool can be used to visualise the data from it, as we have done in Section 6.3.3, where activity profiles for few applications are showed) to make sure that the simulation is indeed accurate (e.g. that the PEs are indeed choosing stealing targets in the way the simulated algorithm prescribes,

that each PE is, at each point in the application execution, either looking for work or executing tasks, that the PEs correctly respond to the stealing requests and so on).

Consider, for example, the execution of the divide-and-conquer application given of Figure 3.2 on the computing environment described in Section 3.3 under Random Stealing algorithm. The following code is the snippet of the trace file generated by SCALES for this execution:

```
...
PE 0 : 2 BLOCK              t:0x1001088e0    (bcnt:2)
PE 0 : 2 TASK_DESCHEDULED   t:0x1001088e0    (gl:0)
PE 0 : 2 TASK_SCHEDULED     t:0x10010c2f0    (gl:0)
PE 0 : 3 BLOCK              t:0x10010c2f0    (bcnt:2)
PE 0 : 3 TASK_DESCHEDULED   t:0x10010c2f0    (gl:0)
PE 0 : 3 TASK_SCHEDULED     t:0x10010de00    (gl:0)
PE 0 : 4 BLOCK              t:0x10010de00    (bcnt:2)
PE 0 : 4 TASK_DESCHEDULED   t:0x10010de00    (gl:0)
PE 0 : 4 TASK_SCHEDULED     t:0x10010e970    (gl:0)
PE 0 : 201 SEND_SCHEDULE    t:0x100108ef0    (d:1)        (sps:2)
PE 0 : 201 TASK_DESCHEDULED t:0x10010e970    (gl:4803)
PE 0 : 201 TASK_SCHEDULED   t:0x10010e970    (gl:4803)
PE 0 : 301 SEND_SCHEDULE    t:0x10010c940    (d:4)        (sps:1)
PE 0 : 301 TASK_DESCHEDULED t:0x10010e970    (gl:4703)
PE 0 : 301 TASK_SCHEDULED   t:0x10010e970    (gl:4703)
PE 1 : 301 SCHEDULE         t:0x100108ef0    20006
...
```

The trace file shows the events where a task is blocked after forking new tasks (`BLOCK`, with bcnt denoting the number of forked tasks), the events where task is scheduled/descheduled (`TASK_SCHEDULED` and `TASK_DESCHEDULED`, with `gl` denoting the remaining size of the current task event, which is `0` for `FORK` events), the events where a task is moved from the spark task queue to the run task queue (`SCHEDULE` event, with the task pointer and the total size of the task), the events where `SCHEDULE` message is sent (with `d` denoting the destination of the message, and `sps` denoting the number of tasks that remain in the PEs spark task queue) and so on. The fourth column in the trace file shows the timestamp of the event.

We have done an extensive testing of the trace files for various small and large applications on small and large computing environments and we concluded that SCALES does indeed accurately simulates the execution of these applications under work steal-

ing. Furthermore, the experiments presented in Chapter 6 show the agreement of results obtained under SCALES with the results obtained using the implementation of the same work-stealing algorithms in Grid-GUM. We can, therefore, conclude that SCALES can indeed be used to approximate the performance of work-stealing algorithms and policies considered in this thesis.

3.5 Grid and Cloud Simulators

Several simulators exist, whose aim is to provide the testbed for implementing different scheduling strategies on heterogeneous distributed computing environments. However, they either do not support the highly dynamic load-balancing mechanisms or they put more restrictions on the applications that they can model than SCALES does.

Perhaps the most widely used simulator for Grid environments is SimGrid (Casanova et al. [CLQ08]). SimGrid represents a framework for evaluating cluster, grid and peer-to-peer algorithms and heuristics. It includes several user interfaces – SimDag for investigation of scheduling heuristics for applications modelled as DAGs, MSG for studying concurrent sequential processes, GRAS which allows users to use SimGrid as a development lab for real distributed applications and SMPI for simulation of MPI applications. The most closely related to SCALES is SimDag. However, it targets the static scheduling algorithms and applications that can be modelled by DAGs. Implementing work-stealing algorithms in it would probably be theoretically possible, but very infeasible, and some of the applications under SCALES model cannot be represented by DAGs.

Another popular Grid simulator is GridSim (Buyya and Murshead [BM02]), whose main goal is to simulate Grid economy. It allows simulation of heterogeneous resources that belong to different administrative domains, and simulations of multiple users executing multiple jobs that compete for these resources. GridSim does not define the application model it uses. The basic unit of application is a sequential task, and it can be parametrised by sequential size (in terms of the number of machine instruction it takes), the size of input and output data and a pointer to its parent task. We can see that this model of task is somewhat simpler that SCALES model, as tasks cannot generate subtasks at different points of their execution. Therefore, not all applications that can be modelled under SCALES can be modelled under GridSim.

MicroGrid (Song et al. [SLJ+00]) is a set of simulation tools which enable Globus applications to be run in arbitrarily virtual grid resource environments. This project focuses on the accurate simulation of computing environments, rather than on the

simulation of different task scheduling/load balancing algorithms. Also, MicroGrid simulates the execution of real applications, rather than application models.

Bricks (Aida et al. [ATN+00]) is another system for the evaluation of different scheduling algorithms for global computing systems. It assumes client/server architecture, where clients submit problems (tasks) that are solved by servers (PEs). Bricks can simulate highly dynamic computing environments, where communication latencies, bandwidths and server load can vary over time. However, it allows modelling of only the bag-of-tasks applications and it assumes that the scheduling algorithms being simulated are centralised (whereas work-stealing is distributed).

Several Cloud simulators also exist. However, they are less relevant to our work, since scheduling in Clouds typically deals with the allocation of individual virtual machines and their balancing over the hosts, rather than on scheduling if individual application tasks. Therefore, we will here just briefly mention some of the most important Cloud simulators.

CloudSim (Calherios et al. [CRB+11]) extends the GridSim simulation toolkit by providing support for modelling and simulation of Cloud-specific features, such as virtualised Cloud-based data centres. CloudSim supports dynamic instantiation of virtual machines, hosts, data centres and applications. It is, however, more oriented towards the Cloud providers, by allowing them to simulate different policies of allocating and scheduling of virtual machines on hosts, rather than towards the developers of runtime systems.

GreenCloud (Kliazovich et al. [KBAK10]) is another Cloud simulator. Its main goal is to capture details of the energy consumed by data centre components in Clouds (servers, switches and links) as well as the details of packet-level communication patterns. It also allows simulation of different workload distribution mechanisms.

3.6 Summary

In this chapter, we presented SCALES, a novel work-stealing simulator, which we use in the rest of the thesis as a testbed for implementation of various work-stealing algorithms and policies. We described the application and computing-environment models that SCALES provide. We gave examples of how SCALES can be used to model divide-and-conquer and data-parallel applications. We also described the mechanism that SCALES uses to simulate the application execution. Finally, we gave an overview of other simulators for heterogeneous distributed computing environments, and compared them to SCALES.

We have seen that SCALES can be used to simulate a wide class of parallel applications and computing environments. It, therefore, enables us to evaluate the algorithms considered in this thesis much more systematically that if only their implementation in a real runtime system is used. It also allows us to obtain reproducible results, which are crucial for the analysis of the performance of algorithms.

Chapter 4

Work-stealing on Heterogeneous Distributed Computing Environments

This chapter describes in more detail the problem we aim to solve, namely, that of efficient load-balancing during the execution of large irregular parallel applications on heterogeneous distributed computing environments. We describe the assumptions we made about applications (Section 4.2), computing environments and runtime systesm (Section 4.3). We also introduce a concept of the *degree of irregularity* of an application (Section 4.2.1), which is used throughout the thesis.

We assume that work-stealing is the method of choice for load-balancing. We describe two important issues that arise in work-stealing in heterogeneous environments: choosing a good target for work stealing request, and responding appropriately to such requests. Finally, we outline our approach to addressing these issues.

4.1 Introduction

Our main objective is to investigate the ways in which efficient load-balancing can be done during the application execution. Good load balancing means that the work is spread evenly across all PEs in an environment, which results in good utilisation and, consequentially, in good speedups of parallel applications. However, achieving a good load balance in the presence of irregularity in parallel applications, coupled with heterogeneity in the underlying computing environment, is a very challenging task. Because parallel tasks vary in size, achieving a good balance of work over all the PEs is not as simple as having each node execute an identical number of tasks.

Also, heterogeneity in communication latencies makes transferring work between some nodes more time-consuming than between others. These are just some of the issues that need to be taken into account when choosing a load-balancing method which will deliver good speedups.

In this chapter, we describe the special class of irregular parallel applications, run-time systems and heterogeneous distributed parallel environments, to be used in this thesis. We then proceed to outline an approach to developing efficient load balancing methods, and compare it to approaches used in less complex setup (e.g. load balancing of parallel applications with regular structures).

4.2 Parallel Applications

The general model of parallel applications considered here is based on the model found in SCALES simulator (see Section 3.2). As a reminder (see Figure 4.1), we are considering applications consisting of a main task, which, during its execution, creates other (possibly parallel) tasks. Each task can be described by a finite sequence of task events, each of which is either a *run* event (where a task runs sequentially for a certain amount of time) or a *fork* event (where a task forks a set of subtasks and waits for all to return their results before moving to the next event).

As seen in Section 3.2, at this level of abstraction, we can model applications that conform to the most popular parallel programming paradigms. Note, however, the restriction that the only data dependencies that exist are between parent and child tasks. Furthermore, data transfers between parent and child tasks happen only before and after execution of child task. Representing applications that have more complex data dependencies (such as, for example, applications that create some kind of pipeline between parallel tasks) is, therefore, not possible with this model. Although this restricts the applications we are considering to those "embarrassingly parallel", we, nevertheless, find our application model general enough to capture important classes of parallel applications. Furthermore, the absence of complex data-dependencies gives us greater flexibility in load-balancing. Under this model, each child task is tied only to its parent task, and can be placed independently of all others. Placing two tasks with many data-dependencies on nodes in different clusters is no longer an issue.

In most of the discussion below, we will restrict our attention to applications that have a simplified, tree-like task structure. In such applications, tasks are of two kinds:

- *nested-parallel tasks*, consisting of three phases (each represented by a single event) – an initial *divide* phase (represented by *run* event), a fork phase (where

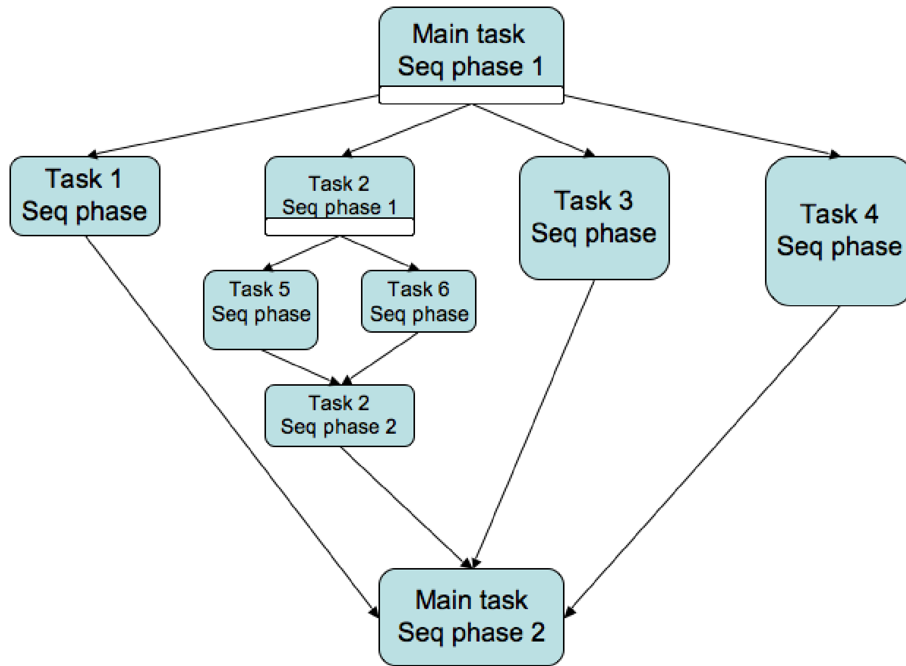


Figure 4.1: Task graph of an example parallel application

a set of subtasks is created, represented by a *fork* event), and the final *conquer* phase (represented by a *run* event), where the results from the subtasks are joined.

- *sequential tasks*, which consist of a single run event.

For each application conforming to this model, we define a set of helper functions that enable us to describe precisely the whole structure of the application. For an application's nested-parallel tasks, we will define functions D and C, which map tasks to integers, and functions F_{seq} and F_{par}, which map tasks to sets of tasks.

For a task t, these functions denote the following:

- $D(t)$ denotes the size (in milliseconds) of its divide phase
- $F_{seq}(t)$ and $F_{par}(t)$ denote sets of sequential and nested-parallel tasks generated in its fork phase
- $C(t)$ denotes the size (in milliseconds) of its conquer phase.

Note that for a task t, either $F_{seq}(t)$ or $F_{par}(t)$ (but not both) can be empty. For each nested-parallel task, we define function F as $F(t) = F_{seq}(t) \cup F_{par}(t)$. For sequential tasks, we define the function S, where $S(t)$ denotes the size (in milliseconds) of sequential task t.

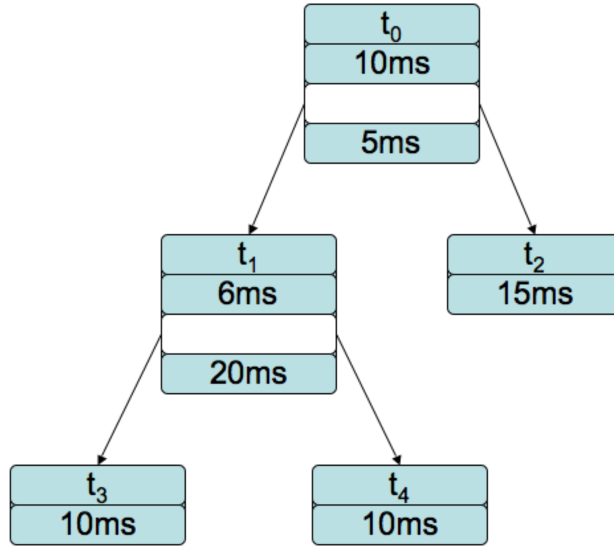


Figure 4.2: Task graph of a simple parallel application

As an example, a task tree of a simple parallel application is shown in Figure 4.2 (t_0 is the main task). For this application, functions D, F, F_{seq}, F_{par}, C and S are the following:

- task t_0 : $F_{seq}(t_0) = \{t_2\}$, $F_{par}(t_0) = \{t_1\}$, $D(t_0) = 10$, $C(t_0) = 5$
- task t_1 : $F_{seq}(t_1) = \emptyset$, $F_{par}(t_1) = \{t_3, t_4\}$, $D(t_1) = 6$, $C(t_1) = 20$
- task t_2 : $S(t_2) = 15$
- task t_3 : $S(t_3) = 10$
- task t_4 : $S(t_4) = 10$

Assuming that t_0 is the main task of an application, we can define the distance of task t to the task t_0 (denoted by $d(t_0, t)$) in the following way:

1. $d(t_0, t_0) = 0$
2. If $d(t_0, t') = k$ and if $t \in F(t')$, then $d(t_0, t) = k + 1$.

If we represent tasks of an application by nodes of a tree, where edges represent the parent-child relationship between tasks, and the main task t_0 is the root of the tree, then $d(t, t_0)$ is the height of node t in the tree.

Since we are dealing only with finite applications, tasks cannot be infinitely nested. Consequently, all tasks that are at a certain distance from the main task must be sequential. For each application, we will denote by T the maximal distance to its main task of any nested-parallel task.

With the appropriate choices of functions D, F_{seq}, F_{par}, C, S and parameter T, we can model various interesting classes of divide-and-conquer and data-parallel applications. We consider these classes below.

Divide-and-conquer Applications

We can see an application conforming to the tree-like task model described above as a divide-and-conquer parallel application, where each task solves some subproblem of an initial problem. The initial problem is solved by the main task. Subproblems are solved either sequentially (in the case of sequential tasks) or they are further divided into subproblems which are solved recursively, in parallel (in the case of nested-parallel tasks). When problems become too small (i.e. after level T in recursion is reached), they are all solved sequentially. Some interesting, more specific examples of divide-and-conquer applications are given in the following:

- **SimpleDC** – Let us make the following assumptions about functions F_{par}, F_{seq}, D, C and S:

  - $|F_{par}(t)| = 2$ and $|F_{seq}(t)| = 0$ for all nested-parallel tasks whose distance from the main task is less than T, and $|F_{par}(t)| = 0$ and $|F_{seq}(t)| = 2$ for all other nested-parallel tasks,
  - for all nested-parallel tasks t, $C(t) = D(t) = K$, where K is a constant close to zero,
  - for all sequential tasks t, $S(t) = C_{seq}$, where C_{seq} is a constant.

  This gives us a simple divide-and-conquer application, where each problem is split into two subproblems, the parts that divide the problem into subproblems and that combine the solutions of subproblems into the solution of the problem are trivial, and all sequential tasks have the same computational cost. We will call this kind of application a *simple divide-and-conquer application*, and denote it by SimpleDC(T, C_{seq}), since the whole application can be described by providing the number of recursion levels and the size of sequential tasks. Figure 4.3 gives the task graph for SimpleDC(3, 10) application.

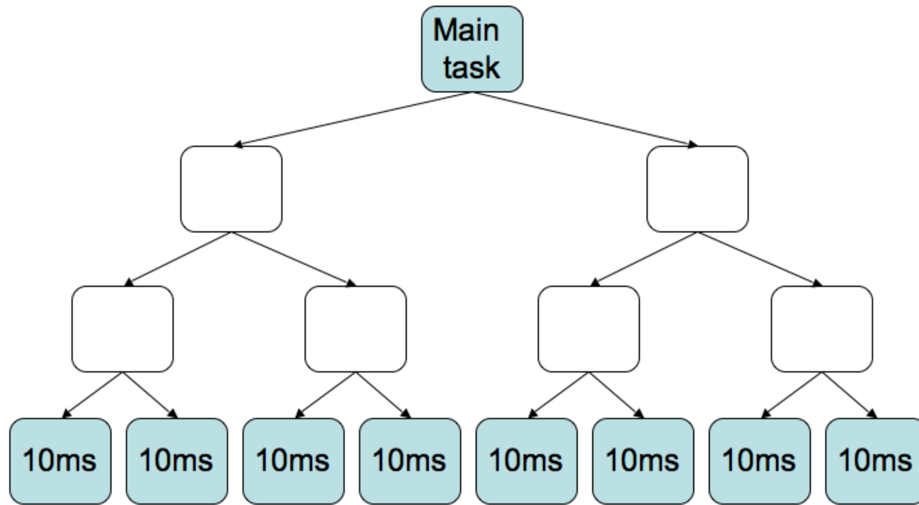


Figure 4.3: Task graph of SimpleDC(T,C_{seq}) application, where $T = 3$ and $C_{seq} = 10$. Nested-parallel tasks are white.

- **DCFixedPar** – Another interesting class of applications can be obtained with the following choices for functions F_{seq}, F_{par}, D, C and S:
  - $|F(t)| = n$, $|F_{par}(t)| = \lfloor \frac{n}{k} \rfloor$ for some $k < n$ (and, consequently, $|F_{seq}(t)| = n - \lfloor \frac{n}{k} \rfloor$) for all nested-parallel tasks t such that the distance from t to the main task is less then T; $|F(t)| = n$, $|F_{par}(t)| = 0$ (and, consequently, $|F_{seq}(t)| = n$) for all other nested-parallel tasks. We also require that, from all tasks generated by a nested-parallel task, every k-th is itself nested parallel;
  - for all nested-parallel tasks t, $C(t) = D(t) = K$, where K is some constant close to zero.
  - for all sequential tasks t, $S(t) = C_{seq}$, where C_{seq} is some constant

  This gives us a class of divide-and-conquer applications where each nested-parallel task generates the same number of tasks, and, if its distance from the main task is less than T, has the same ratio between sequential and nested-parallel tasks it generates. We can see SimpleDC applications as instances of this class of applications, with $n = 2$, $k = 1$. We will call this kind of applications the *divide-and-conquer applications with fixed amount of parallelism*, and denote it by DCFixedPar(n, k, C_{seq}, T). See Figure 4.4 for a task graph of an example of DCFixedPar application, where $n = 6$, $k = 3$ and $C_{seq} = 5ms$.

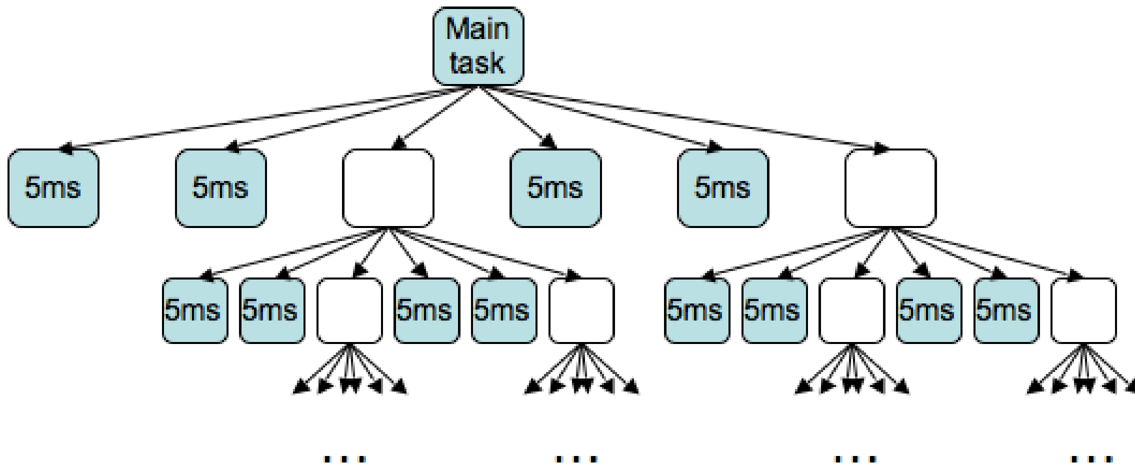


Figure 4.4: Task graph of DCFixedPar(n, k, C_{seq}, T) application, where $n = 6$, $k = 3$, $C_{seq} = 5ms$. Nested-parallel tasks are white.

An interesting classification of divide-and-conquer applications has been given by Herrmann [Her01], who describes several variations of the general divide-and-conquer pattern. These enable various compiling/runtime optimisations (such as more efficient static task placement). Herrmann describes six hierarchical classes of the divide-and-conquer applications. We will here describe the first three classes in the hierarchy, since they describe the structure of the tasks structure of the applications:

1. *General DC* – the most general class of divide-and-conquer applications, which covers all applications that decompose the initial problem into subproblems, solve them independently and then combine the solutions. The only requirement for this class is that the subproblems are independant (i.e. they do not have any data-dependencies). Examples of this class include the Quicksort algorithm and the Maximum independent set algorithm for finding the largest set of independent nodes in a graph.

2. *DC with static schedule* – applications that belong to the General DC class and where the number of recursion levels is known beforehand (i.e. where the recursion level can be passed as a parameter to the application). An example is the n-queens problem.

3. *DC with a static schedule and allocation* – in addition to the conditions for the DC with static schedule class, here the number of subproblems generated at each level is known in advance. An example is the Mergesort algorithm.

The remaining three classes consider ways in which the division of the data is done in the divide phase of the divide-and-conquer applications. Since we ignore the issues of data divisions and transfers, these classes are of not directly relevant to us. We can see that SimpleDC and DCFixedPar classes of applications that we have defined here belong to the third class, i.e. DC with a static schedule and allocation class.

Data-parallel applications

Another way of looking at an application with a tree-like structure of tasks is to see it as a data-parallel application, which potentially has nested parallelism in it. If T (the maximal distance of any nested-parallel task to the main task) is 0, we get a simple data-parallel application without nested parallelism.

For $T > 0$, we get an application with nested parallelism, with a similar structure as DCFixedPar applications, but where the number of sequential and nested-parallel subtasks is not necessarily the same for each nested-parallel task. This kind of parallelism arises in, for example, a data-parallel application which maps some function over some non-linear data structure (for example, a tree). For some elements of the data structure, the function is evaluated sequentially (sequential tasks), but for others, mapping a function over them generates further data-parallelism. One example of such an application is the parallel implementation of the Min-Max algorithm, where a tree of game positions is processed (by assigning a value to each position). For each position, the positions that can follow it after the player's move are evaluated in parallel, until some depth of the game tree is reached, where the remaining positions are evaluated sequentially. Sequential tasks also correspond to the positions after which the tree is pruned (and, therefore, no further parallelism is generated). Also, Monte Carlo Photon Transport algorithm (Hammes and Bohm [HB95]) is an example of application with such parallelism.

Specifically, in Chapter 6 we will be interested in single-level data-parallel applications, where task sizes are generated randomly under normal distribution with the mean value m and standard deviation d. We will denote such an application, with n tasks, as SimpleDataPar(n,m,d).

4.2.1 The Degree of Irregularity of Parallel Applications

In Section 1.3, we have described what does it mean for an application to be irregular, and we have pointed out that we focus on *cost-irregular* parallel applications (i.e. those applications whose parallel tasks have either different sizes, different patterns of

communications or they generate different amounts of parallelism).

Since in our model of parallel applications, communication patterns for all tasks are the same (i.e. every task communicates just with its parent task, and the amount of communication is the same for all tasks), we can see that irregularity can either arise from different tasks having different sizes or generating the different amount of parallelism. Here, by the *size of the task* we mean the computational cost (measured, for example, in milliseconds or in the number of machine instructions) of its sequential execution (including all of its subtasks), and by the *amount of parallelism* it generates we mean the number of tasks it generates in its fork phase (0 for sequential tasks). We will focus on the irregularity in task sizes, since if the nested-parallel tasks of an application generate different number of tasks, then it is also likely that the sizes of these nested-parallel tasks will be different. Therefore, we can see a class of applications which have irregularity in task sizes as a superset of those that have the irregularity in the amount of parallelism they generate.

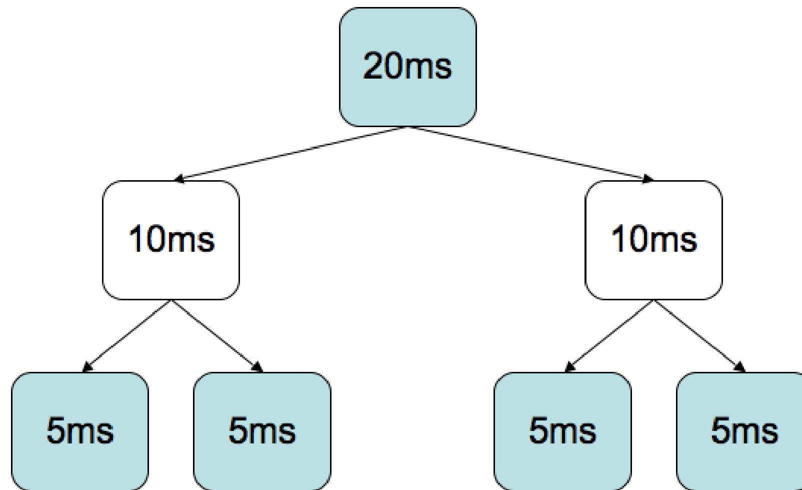
In the following, we will precisely define the size of a task for applications we consider in this thesis. Assume that the task t is represented as a sequence of events $(e_1, e_2, \ldots, e_n)$, whereas each e_i is either RUN $r(e_i)$ or FORK $(t_{i_1}, t_{i_2}, \ldots, t_{i_{n_i}})$, where $r(e_i)$ is integer and $t_{i_1}, t_{i_2}, \ldots, t_{i_{n_i}}$ are tasks. Then the size of such task is defined in the following way

Definition 1 (The size of a task) *For a task $t = (e_1, e_2, \ldots, e_n)$, where e_i is task event, its size ($Sz(t)$) is defined by*

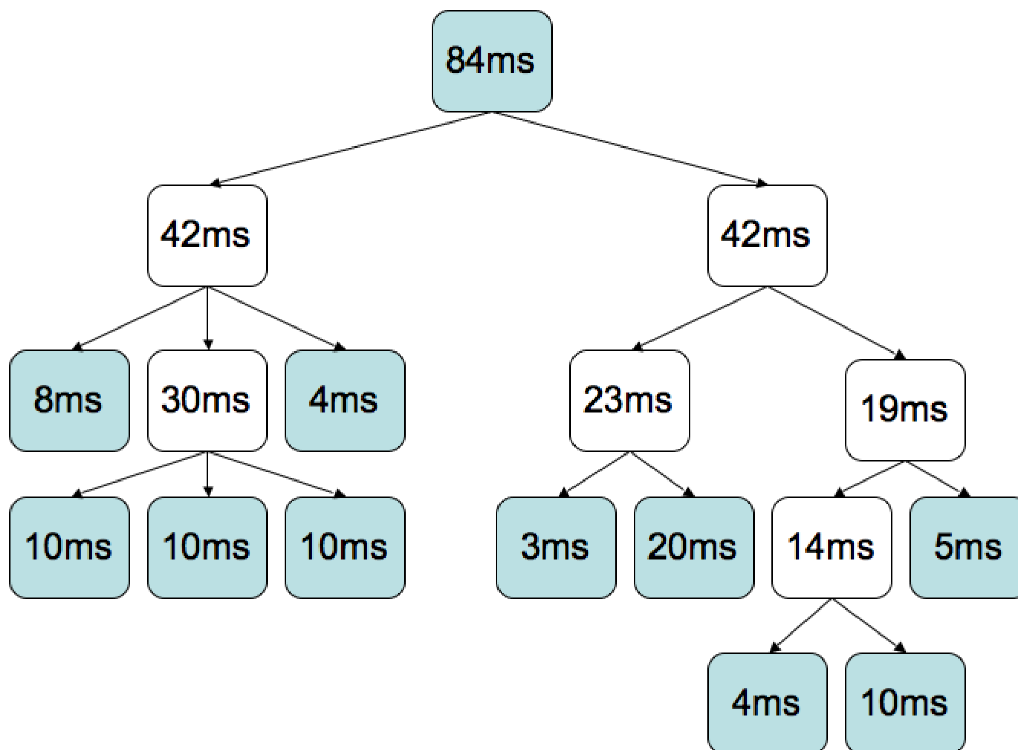
1. $Sz(t) = \sum_{i=1}^{n} SzEvent(e_i)$
2. *The size of an event e is defined by*
   (a) *If e is a run event, i.e. $e = RUN\, x$, then $SzEvent(e) = x$*
   (b) *If e is a fork event, i.e. $e = FORK\ (t_1, t_2, \ldots, t_k)$ where $t_1, t_2, \ldots, t_k$ are tasks, then*

$$SzEvent(e) = \sum_{i=1}^{k} Sz(t_i).$$

On Figure 4.5, we can see an example task trees of two parallel applications, where tasks are annotated by their size. Task represented by blue squares (except for the main task) are sequential, whereas the ones represented by white circles are nested parallel, consisting of just one FORK event, where a set of subtasks that are its children in the tree are generated. We can see that the size of a nested-parallel task is aggregate size of all of the subtasks it generates.



(a) SimpleDC application



(b) Irregular application

Figure 4.5: Task graphs of two simple parallel applications

Having the definition of the size of a task in hand, we can now define the *degree of irregularity*, the notion which tell us exactly “how different” the sizes of parallel tasks that comprise an application are. Our goal is for the definition of the degree of application irregularity to comply with the intuitive notion of the “amount” of irregularity, where “more irregular” applications have tasks of more variable size and structure.

We will start with applications that do not have nested parallelism. In these applications, the main task generates a number of sequential child tasks. Assuming that we denote the mean size of these child tasks by m, it seems natural to define the *degree of irregularity* of such an application as a standard deviation of sizes of child tasks from m. Regular application will have tasks of similar size, therefore their deviation from m we be relatively small. Highly irregular application will have tasks whose sizes differ dramatically, so their deviation from m will be larger.

However, taking only standard deviation from the mean task size as a measure of irregularity quickly gets us into problems. For example, if two applications, one with mean task size of 10ms, and the other with mean task size of 300ms, both have the standard deviation of 5ms, then most of the tasks from the first application will have sizes between 5ms and 15ms (50% difference to the mean task size), and for the second application, most sizes will be between 295ms and 305ms (1.6% difference from the mean task size). Under the above definition, these two application would have the same degree of irregularity, although clearly the first one is much ’more irregular’, since task sizes have much higher amplitude with the respect to the mean task size. Therefore, we will define the degree of irregularity of an application with a single-level of parallelism in the following way

Definition 2 (Degree of irregularity of application without nested parallelism) *For parallel application A without nested parallelism, let $mean_{size}(A)$ denote the mean size of tasks generated by the main task, and let $\delta_{size}(A)$ denote the standard deviation of sizes of these tasks from $mean_{size}(A)$. That is, if $t_1, t_2, \ldots, t_n$ are all subtasks of the main task, then*

$$mean_{size}(A) = \frac{\sum_{i=1}^{n} Sz(t_i)}{n}$$

and

$$\delta_{size}(A) = \sqrt{\frac{\sum_{i=1}^{n} |Sz(t_i) - mean_{size}(A)|^2}{n}}.$$

Then the degree of irregularity *of the application A, denoted by $irr(A)$, is defined as*

$$irrSize(A) = \frac{\delta_{size}(A)}{mean_{size}(A)}.$$

Note that, in statistical terms, the degree of irregularity of an application is really a *coefficient of variation* of the task sizes. This measure tells us how different the task sizes of the application are. Since all of the task sizes are positive, coefficient of variation is in this case the same as a *relative standard deviation* (which is defined as $|\frac{\delta}{\text{mean}}|$). Another similar statistical measure is *variance-to-mean* ratio (defined as $\frac{\delta^2}{\text{mean}}$). However, nothing would particularly change if we used this ratio – if the application A has smaller degree of irregularity than the application B under our definition, it will also have smaller degree of irregularity if we use variance-to-mean ratio as the measure of irregularity.

We now turn our attention to applications with nested parallelism. Using the same definition of the degree of irregularity as for applications without nested parallelism would mean that the degree of irregularity does not depend on the structure of subtasks generated by the main task, but only on their sizes. Therefore, for example, two applications on Figure 4.5 would have the same degree of irregularity, although intuitively the application of Figure 4.5b is more irregular. Furthermore, the degree of irregularity of both applications would be 0. On the other hand, taking the standard deviation from the mean size of *all* of application's tasks (where the sizes of sequential tasks are accounted for also in the sizes of their parent nested-parallel tasks) would make even simple applications, like the one on Figure 4.5a have a high degree of irregularity. For that application, task sizes are 20ms, 10ms, 10ms, 5ms, 5ms, the mean task size would be 8.57ms, deviations of task sizes from this main task size would be 11.43ms, 1.43ms, 1.43m, 3.57ms, 3.57ms, making the standard deviation 5.66ms, and, therefore, the degree of irregularity 0.66. Similar SimpleDC applications with more tasks (for example, the one on Figure 4.3) would have even higher degree of irregularity, although their structure is identical to the one on Figure 4.5a. Intuitively we would not call these applications irregular, since each nested-parallel tasks generates subtasks of identical structure and size.

To circumvent the problems mentioned above, we will adopt the definition of the degree of irregularity of a parallel application that takes into account the irregularity of its individual nested-parallel tasks (including the main task). We first define the *degree of irregularity of a nested-parallel task* (the definition very similar to the degree of irregularity of an application without nested parallelism), which will tell us

how different the subtasks generated by that task are. Then, the overall degree of irregularity of a parallel application will be the mean degree of irregularity of all of its nested-parallel tasks, which shows how irregular these tasks are on the average.

Definition 3 (The degree of irregularity of a nested-parallel task) *Let $t_1, t_2, \ldots, t_n$ be all tasks generated by a nested-parallel task t in an application. Let $mean_{size}(t)$ denote the mean size of tasks $t_1, t_2, \ldots, t_n$ and let $\delta_{size}(t)$ denote the standard deviation of sizes of these tasks from $mean_{size}(t)$. That is,*

$$mean_{size}(t) = \frac{\sum_{i=1}^{n} Sz(t_i)}{n}$$

and

$$\delta_{size}(t) = \sqrt{\frac{\sum_{i=1}^{n} |Sz(t_i) - mean_{size}(t)|^2}{n}}.$$

Then the degree of irregularity *of the task t, denoted by $irr(t)$, is defined as*

$$irr(t) = \frac{\delta_{size}(t)}{mean_{size}(t)}.$$

Definition 4 (The degree of irregularity of an application) *Let $tn_1, tn_2, \ldots, tn_n$ be the set of all nested-parallel tasks (including the main task) of the application A. Then we define the* degree of irregularity *A, denoted by $irr(A)$, as*

$$irr(A) = \frac{\sum_{i=1}^{n} irr(tn_i)}{n}$$

We can see that the definition of the degree of irregularity of application without nested parallelism conform to this more general definition, since in this kind of applications the only nested-parallel task is the main task, and the degree of irregularity of the application is the degree of irregularity of its main task.

This definition conforms to the intuitive notion of irregular applications. For example, with that definition, the SimpleDC applications have the degree of irregularity of 0, since every nested-parallel task generates the subtasks of an identical size. Consider now the application on Figure 4.5b. We will consider the degree of irregularity of each nested-parallel task separately:

- The main task has two subtasks, both with sizes $42ms$. Therefore, all of its subtasks are of equal size, so its degree of irregularity is 0.0.
- Task t_1 has subtasks t_3, t_4 and t_5 with sizes $8ms, 30ms$ and $4ms$ respectively. Therefore, $mean_{size}(t_1) = 42ms/3 = 14ms$. The deviations from the mean task

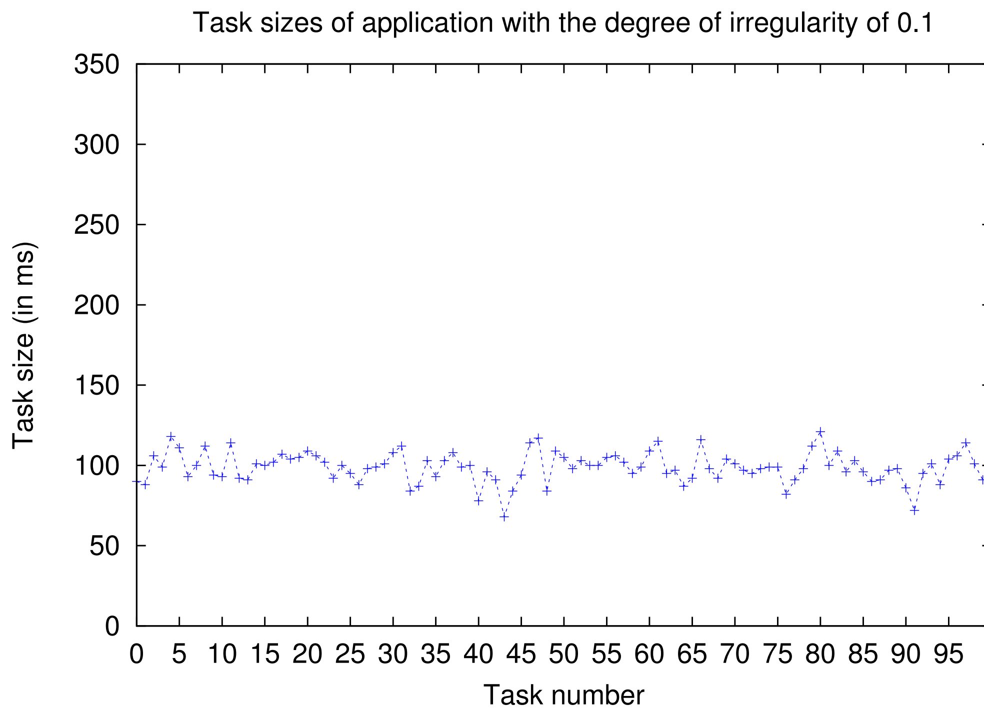
size of tasks t_3, t_4 and t_5 are $6ms$, $16ms$ and $10ms$ respectively, making the mean deviation $\delta_{size}(t_1) = \sqrt{\frac{130.66}{3}} = 11.43ms$ Therefore, the degree of irregularity of task t_1 is $irr(t_1) = 11.43/14 = 0.81$.

- Similarly, for tasks t_2, t_4, t_6, t_7, t_{13} the degrees of irregularity are 0.09, 0, 0.73, 0.47 and 0.42, respectively.

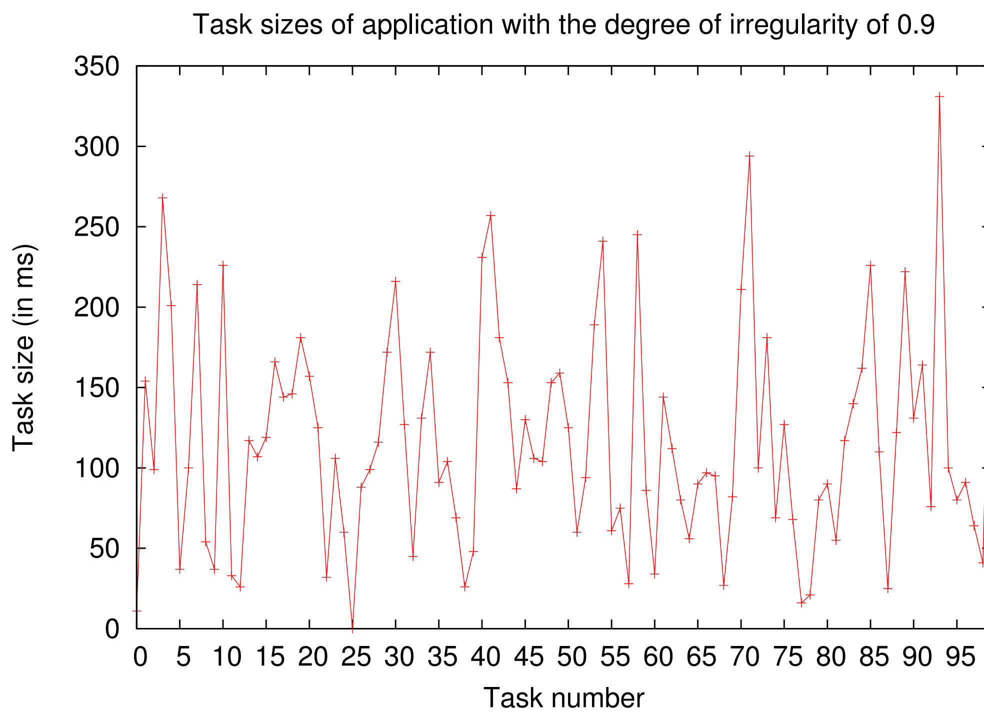
Taking the average of the degrees of irregularity of individual nested-parallel tasks, we get the overall degree of irregularity of an application to be 0.36. This means that, although the application is irregular, it is not "too" irregular, since some of its nested-parallel tasks are very regular.

We will now give a few more examples of the degrees of irregularity of some kinds of applications we will consider in this thesis. Figure 4.6 gives the graphs of task sizes of applications without nested parallelism, each comprising 100 tasks, with the mean task size of 100ms. Sizes of application tasks are generated randomly, with normal distribution. Applications have size-irregularity levels of 0.1 and 0.9. We can see that the sizes of tasks for more irregular applications are much more variable than in the case of more regular one. This kind of application, where the main task generates a set of sequential subtasks, sizes of which are generated randomly with normal distribution will be the focus of Chapter 6. We will denote the class of applications with the mean task size of m and the degree of irregularity i, where the main task generates t sequential subtasks, by $SingleDataPar(t, m, i)$.

Table 4.7 gives a few examples of the degrees of irregularity for DCFixedPar applications. We can observe that for DCFixedPar(n, k, C_{seq}, T) applications with a fixed value for n, if we increase k (that is, if nested-parallel tasks get sparser), we get more irregular parallel application. This is the main motivation for introducing DC-FixedPar applications – we want to observe the performance of various load-balancing algorithms for applications with nested-parallelism, which have the increasing degree of irregularity. We can also see the degree of irregularity of DCFixedPar applications as a measure of how balanced their task tree is. For very regular applications (for example, DCFixedPar(2,1,C_{seq},T), which is the same as SimpleDC(T,C_{seq})), the task tree is balanced. The more irregular the application is, the more unbalanced its task tree is.



(a) i=0.1



(b) i=0.9

Figure 4.6: Graphs of task sizes of the example $SingleDataPar(100, 100, i)$ applications

Application	**The degree of irregularity**
DCFixedPar$(10, 1, C_{seq}, D, C, 4)$	0.00
DCFixedPar$(10, 2, C_{seq}, D, C, 4)$	0.163071
DCFixedPar$(10, 3, C_{seq}, D, C, 4)$	0.364778
DCFixedPar$(10, 4, C_{seq}, D, C, 4)$	0.608418
DCFixedPar$(10, 6, C_{seq}, D, C, 4)$	1.116541

Figure 4.7: The degrees of irregularity for various DCFixedPar applications

4.3 Heterogeneous Distributed Computing Environments and Runtime systems

In the heterogeneous distributed computing environments that we consider, heterogeneity can come from different sources:

- Different PEs can have different computing capabilities.
- Different amounts of memory can be associated to different PEs.
- Memory associated to a PE can itself be heterogeneous, in terms of different latencies in access to different portions of it (e.g. access to the L1 cache is much faster than access to the L2 cache, which is in turn much faster than access to the main memory).
- Different network links can have different bandwidths and communication latencies.

In this thesis, however, we focus solely on the *communication latency heterogeneity* – that is, heterogeneity in latencies in network communication between different PEs.

We assume that the distributed computing environment is structured as a group of interconnected clusters of PEs (Figure 4.8), where the communication latency between PEs in a single cluster is both uniform and much shorter than that for PEs in different clusters. We also assume that the latency between two fixed PEs is constant, and that the bandwidth is infinite. We further assume that all PEs have identical computing capability and that the amount of memory associated to all PEs is equal. Additionally, we assume that the memory associated with each PE is uniform – a PE can access each and every part of its memory in the same amount of time. That is, we define a *computationally uniform* computing environment as a special class of heterogeneous distributed computing environment that is *heterogeneous* in terms of communication latency, but which is *homogeneous* in terms of the computing capability and memory

of each PE. This allows us to focus purely on the problem of *dynamic load balancing in the presence of heterogeneous communication latencies*, without also needing to consider relative PE performance and cost of accessing different hierarchical levels of memory.

Concerning the runtime systems that we consider, besides the restrictions that we have already made, namely, that runtime system is dedicated to the execution of a single parallel application at time, and that the work-stealing is used as a method of load balancing, we make one more important restriction with respect to the task execution. Namely, we will assume that the migration of tasks under execution is not possible.

We differentiate between the tasks whose execution has not already started and the tasks under execution. The former ones are represented by some sort of task descriptor which encapsulates enough information to start the task execution (for example, the name and the arguments of the function that the task executes), and are typically much more lightweight than the latter ones, which need to keep the information about their whole state (including registers and stack). We will assume that only the tasks whose execution has not started can be transferred between PEs. Therefore, in the rest of the thesis, when we say that task is *transferred* from PE A to PE B, we mean that just the task descriptor (created when the task is forked by its parent task) is transferred from A's to B's task pool.

Note that in Grid-GUM, sparks play the role of task descriptors. Sparks are eventually either discarded or converted into threads, which play the role of tasks in our discussion. Grid-GUM also supports the mechanism of thread migration, where the whole state of the thread under evaluation can be migrated between PEs. However, since the cost of doing this is nontrivial and depends on the thread profile, and since we are interested in results applicable to other runtime systems as well (which may not support this feature), we will assume that migration of tasks under execution is not possible.

4.4 Work-stealing on Heterogeneous Computing Environments

In Section 2.3.1 we reviewed the current state-of-the-art work-stealing algorithms for heterogeneous distributed computing environments. As we have seen, most of these algorithms are designed to deal with divide-and-conquer parallel applications, and their aim is to somehow hide the costs of wide-area (WAN) communication. Some algo-

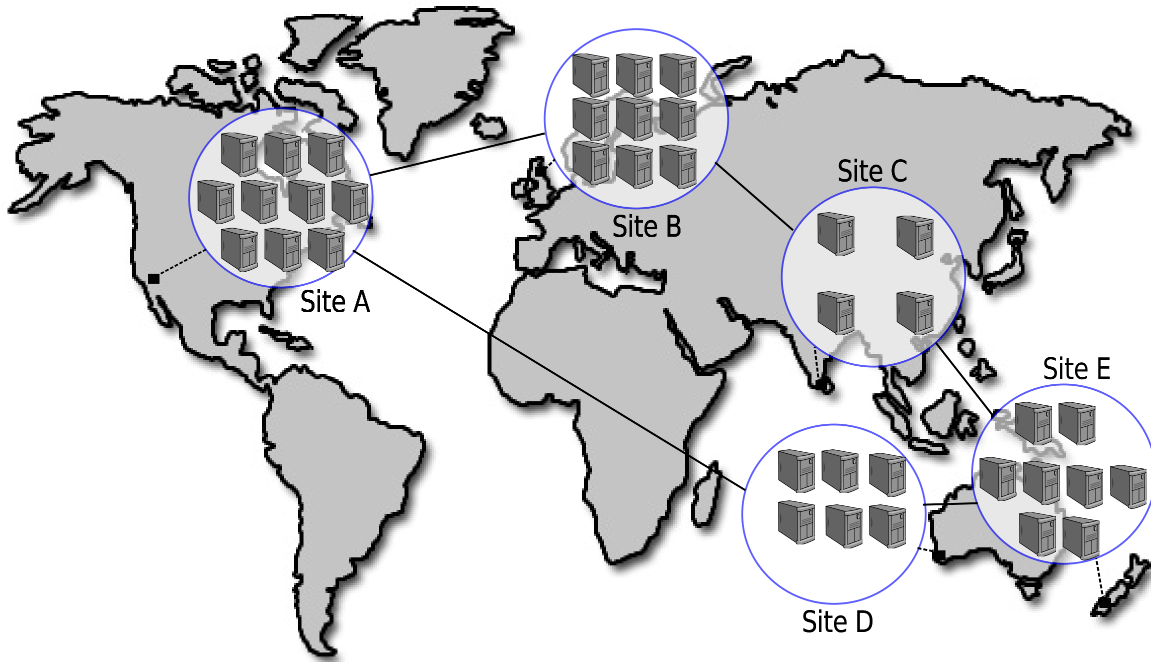


Figure 4.8: Grid connecting 5 sites from Europe, North America, Asia, Australia and New Zealand

rithms (e.g. Hierarchical Stealing) aim to optimise WAN communication by sending stealing-related messages over WAN only when there is no work that could be stolen over local-area network (LAN). Others, such as (A)CRS, aim to overlap WAN and LAN communication, by performing remote work prefetching in parallel with local work stealing. The common denominator for all these algorithms is that they address only one issue that arises during the work stealing - *how should a thief choose target(s) for its work stealing attempts*, and they do it by preferring stealing over LAN to that over WAN. Furthermore, most of them address this issue by taking into account only static information about network topology (e.g. communication latencies between PEs), and not runtime information about the PE load. Also, by assuming some common characteristics of divide-and-conquer applications (e.g. tasks created earlier in the application execution are larger and induce more parallelism), they address some other, equally important issues, such as *what task should the victim send as a response to the thief* in more or less ad-hoc way, by always offloading the *oldest* task from the victim's task pool.

The approach that mentioned algorithms take is sensible for simpler divide-and-conquer applications (e.g. SimpleDC). Since most of the tasks generated in such applications induce a lot of parallelism, if one PE from a cluster succeeds in stealing

task, that task can keep all PEs from that cluster busy for a long time. Therefore, we do not expect that PEs will have to look for work over WAN too often, and we also expect that most of the PEs will have some extra work to offload. Due to this, the probability of thief finding a target with work to offload is pretty high even if the target is chosen randomly, without trying to use the information about current PE loads. Also, by always offloading the oldest task in victim's task pool as a response to steal attempt, we make sure that we send the thief the task that will create the most work. This means that we answer the question of *what* task to send to the thief implicitly by always offloading the task that creates the most additional parallel work. If a PE needs to execute one of its own tasks, it executes the *newest* one. This preserves locality, by executing tasks near where the data they need resides (assuming that the data for most recently created task is still near the PE that has created it, perhaps in its cache memory).

If we consider irregular parallel applications (for example, DCFixedPar applications, especially if function S is not constant) the situation is not that simple any more. There, the age of the task might be unrelated to its size and the amount of parallelism it creates. Some tasks may be sequential, and some may generate a lot of parallelism. Also, different sequential tasks may have different sizes. This means that not all of the PEs in an environment might have work to offload to a potential thief, and some of them might only have very small, sequential tasks in their task pool. Therefore, choosing the target for a steal attempt randomly, without using any load information, might not be viable any more. Also, in some cases, locality might not be much of an issue so it might be perfectly fine to execute some task other than the newest one locally. All of this means that runtime systems that support the execution of irregular parallel applications need to consider different policies for locating work and choosing which work to send as a response to steal attempt than systems that support only divide-and-conquer applications.

One additional drawback of the algorithms considered above is that the victim treats all steal requests in the same way, no matter where they come from (i.e. it chooses the same task for offloading regardless of how far does the steal request come from). In other words, information about the network topology is only used for locating work, and not for deciding on how to respond to steal attempt. In heterogeneous environments, the difference in communication latencies between PEs that are nearby (e.g. on the same multicore machine or within the same cluster) and distant (e.g. that belong to different clusters) can be very high. In this situation, the victim might use a different strategy when responding to a steal attempt from a 'closer' PE than from

a further one. It may be better to send smaller tasks to PEs that are close, in order to save the large ones for more distant PEs. Alternatively, in some other situation, we may want to avoid offloading tasks to distant PEs altogether, for example if closer ones can execute them faster.

In the reminder of the thesis, we investigate how to find answers to the two crucial questions that arise in work stealing on heterogeneous distributed systems:

- How should a thief choose the appropriate steal target?
- How should the victim respond to steal attempts that it receives (i.e. how many and what tasks should it send as a response)?

We aim to find the answers that are applicable to a wide class of applications (rather than just to simple divide-and-conquer ones). More details of our approach to answering the two posed questions are given in the following two sections.

4.4.1 How to Choose Steal Targets

In its simplest form, work-stealing is a fully decentralised mechanism, since each PE makes load-balancing decisions independently of all other PEs. However, this also means that a thief do not have any information about load of a target from which it attempts to steal. We have already argued that, in irregular parallel applications, only a small number of PEs might have extra work, and the thieves might struggle to find them if they do not have any information about the system load. This is especially an issue in large-scale computing environments. In order to find a good answer to the question of *how to choose steal targets*, we, therefore, argue that it is necessary to use some form of *dynamic PE load* information.

Having decided to use the dynamic PE load information, two important issues arise. Firstly, what is the best way to use this information? In other words, assuming that a thief knows the load of every possible target in the system, what specific target should it choose for stealing? Is it the best to choose a *random* target with work, the *nearest* one, or maybe the one with the *highest-load*? Alternatively, is it worthwhile to do local and remote stealing in parallel, as is done in Cluster-Aware Random stealing? We explore this issue by extending the state-of-the-art work-stealing algorithms described in Section 2.3.1 with the *perfect load information*. That is, using simulations, we consider the hypothetical scenario where thieves under these algorithms know the loads of all other PEs, and use this information in selecting the steal targets. In Hierarchical work-stealing, for example, a thief would avoid looking for work in its subtree if it

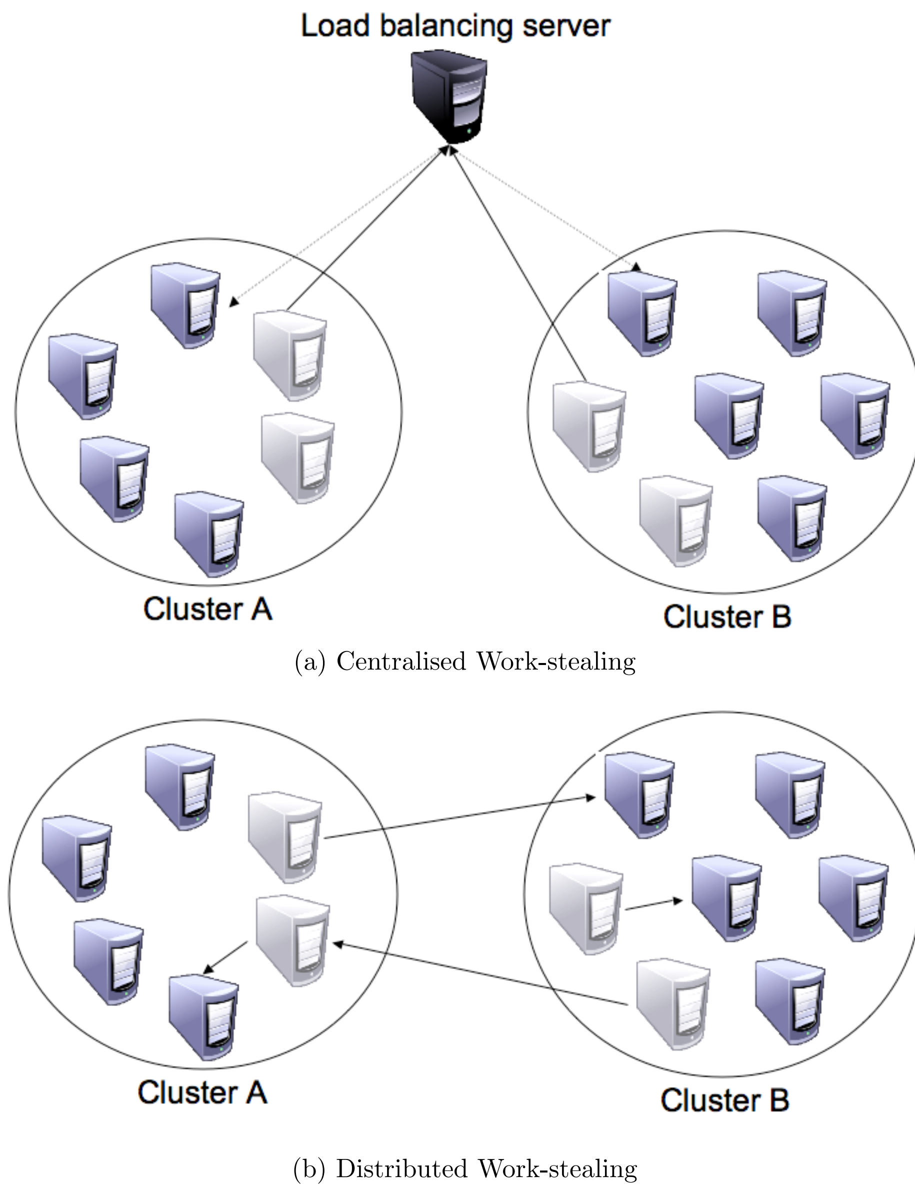
knows that no PE there has any tasks to offload. It would, instead, immediately ask its parent for work. As another example, In Random stealing, a thief would restrict the set of possible targets (one of which is chosen randomly) to only those PEs with work.

Once we discover what is the best way of using load information, we need to investigate the question of how to obtain it. That is, we need to find a way for PEs to obtain the accurate load information during the application execution.

Grid-GUM tries to solve this problem by attaching load information to the steal messages exchanged during the application execution (see Section 2.4.3 for more details). Initially, each PE knows only its own load. However, as steal messages are exchanged, PEs get the information about loads of other PEs. Eventually, each PE should have accurate information about the load of the rest of the system, and it should know where to look for work once it becomes idle. However, this approach has several drawbacks. Firstly, the size of each steal message is increased with the information about the load of every PE in the environment. In large computing environments, this might make steal messages prohibitively large. Secondly, the assumption that each PE can have accurate information about the load of every other PE in the environment is overly optimistic. The accuracy of load information that a PE has depends on how often does it communicate with other PEs. An isolated PE (i.e. the one that does not communicate with the rest of the system too often, perhaps because the rest of the system assumes that it has no work to offload) will have inaccurate information about the load of the rest of the system, and, conversely, the rest of the system will have inaccurate information about its load. This PE might be heavily loaded, and an ideal victim from which to steal work, but other PEs just will not try to steal from it, as they assume that it has no work.

In order to address the problems with Grid-GUM work stealing, we can try a *fully-Centralised* approach to keeping the load information (See Figure 4.9). This would mean that one PE is nominated as a central load-balancing PE, and that all PEs would periodically send the information about their load to it. Additionally, each thief would contact the load-balancing PE in order to find out what PEs would be suitable targets for stealing. This would enable load-balancing PE to have accurate information about the load of other PEs, and thieves to locate victims more easily. However, in a heterogeneous distributed computing environment, this approach is problematic for a number of reasons, for example:

- The presence of heterogeneous and, potentially, very high communication latencies makes collecting the information about the load of PEs very expensive.



(a) Centralised Work-stealing

(b) Distributed Work-stealing

Figure 4.9: Centralised and distributed work-stealing. Grey PEs are idle, purple ones are busy and the black PE represent the dedicated load balancing node.

- Load-balancing PE can become a bottleneck, due to a fact that possibly very large number of PEs will need to communicate with it relatively frequently.

- Generally, individual PEs in distributed environments are unreliable, and may go offline frequently. If one of the load balancing nodes goes offline, then some mechanism that will propagate its role (together with its information about the system load) to some other node is necessary.

We propose a novel approach which combines fully-distributed and fully-Centralised approach to exchanging the load information between PEs. We use a fully-Centralised approach within low-latency networks, where PEs from the same cluster send their load information to a central load-balancing PE at regular time intervals. In this case, the central PE will have accurate information about the load of its cluster. Over the high-latency networks, however, we use a fully-distributed approach, where load information is attached to steal messages that are sent anyway (similarly to Grid-GUM), so the exchange of load information over high-latency networks does not incur any overhead. In this way, we are hoping to have benefits of both approaches while circumventing their major drawbacks.

4.4.2 How to Respond to Steal Attempt

To answer the question of how victims should respond to steal attempts they receive, we use the information about the sizes of tasks in victims' task queues to drive the decision of what task to send to what thief. Rather than using the implicit assumptions about characteristics of parallel tasks (such as that the age of the task corresponds to the size and the amount of parallelism it generates), as is done in the most of the state-of-the-art work-stealing algorithms, we assume that explicit and exact information about the size of each of the application's tasks is available to the runtime system. That is, we assume that we have a *perfect* knowledge about parallel profiles of all tasks that might be created at any point in application execution, and we are concerned with how this knowledge can be exploited. In the thesis, we are not addressing the question of how can this perfect information be obtained.

Our goal when finding the answer to the posed question is to explore we explore the different *task selection policies* that use the task granularity information. We can see these policies as functions which take as an input the set of tasks in victim's task queue (together with their parallel profiles), the distance (in terms of communication latency) between the thief and the victim and possibly some information about system

load, and as a result return a task from the task queue that should be sent from the victim to the thief.

Although this question is related to the one we considered in the previous section, we consider it independently, since the results we obtain in answering to this question are independent of the way in which PE chooses its steal target. This makes results obtained in investigating this question applicable to many different work-stealing algorithms and, therefore, different runtime systems.

4.5 Summary

Since the general problem of efficient work-stealing for irregular parallel applications on heterogeneous distributed computing environments is rather broad, in this chapter we made some simplifying assumptions about the type of the applications we consider (restricting the patterns of communication that can exist between application's tasks), heterogeneous environments on which they are executed (by assuming the uniformity in PE processing powers, memory associated with them and bandwidth in links between them) and runtime systems that act as middlewares between applications and computing environment (by restricting the task migration to tasks whose execution still has not started).

In the description of applications that we consider, we have defined a key concept of the degree of application's irregularity. Using this concepts, rather than just classifying applications as either 'regular' or 'irregular', we are able to tell exactly *how much* (i.e. to what degree) irregular the application is. This enables us to compare two applications quantitatively and determine which is *more* irregular. This will further enable us to establish relationships between the performance of various work-stealing algorithms and the irregularity of applications.

Having described the computing environments, applications and runtime systems we consider in this thesis, we subsequently focused on two important questions that are related to work-stealing in our settings (i.e. finding a good method of choosing steal targets and a good method of responding to steal attempts). We pointed out how state-of-the-art work-stealing algorithms do not address these questions properly if the applications that are executed are irregular, and outlined our approach in exploring the answers to them.

Each of the next two chapters deals with one of the two question we aim to answer. Chapter 5 explores methods for choosing good targets for steal attempts. Our approach is to use the dynamic PE load information, and to combine the Centralised

and distributed methods of obtaining this information. Chapter 6 explores methods of choosing what tasks should a victim offload as a response to steal attempt from thief, by taking advantage of perfect information about the sizes of application's tasks.

Chapter 5

Load-Based Topology-Aware Work Stealing for Heterogeneous Distributed Computing Environments

In this chapter, we investigate the use of information about PE loads in work-stealing. We use a standard definition of a *PE load*, where it is defined as the number of tasks in PE's task pool. In Section 5.1, we investigate how load information can be *used* to guide the decisions of choosing the steal targets. That is, we propose the algorithms that a thief can use to choose an appropriate target, under the assumption that it has an accurate information about the load of every PE in the computing environment. In Section 5.2, we evaluate which of the proposed algorithms gives the best speedup for a wide class of applications on many homogeneous and heterogeneous computing environments. We also evaluate how much individual state-of-the-art work-stealing algorithms, described in Section 2.3.1, could benefit from the presence of load information.

After showing *how to use* the load information, we turn our attention to investigating the question of *how to obtain it*. In Section 5.3, we present a novel *Feudal Stealing* algorithm, which is based on the Cluster-Aware Random Stealing (CRS) algorithm, and which uses a combination of centralised and distributed methods of exchanging information about the loads of individual PEs in order to obtain a good approximation of the load of the whole environment. In Section 5.4, we evaluate the Feudal Stealing algorithm against Grid-GUM, which uses a fully distributed way of exchanging load information, and also against the CRS and Hierarchical Stealing algorithms.

In [VNKB01], Van Nieuwpoort et al. have compared the CRS algorithm with other algorithms that we consider, and that do not use load information, on a small set of regular divide-and-conquer applications, and their conclusion was that this algorithm gives the best speedups in (almost) all of the cases. Our experiments have confirmed their results. Furthermore, for irregular parallel applications, we discovered that either (A)CRS or Hierarchial Stealing give the best speedups for all applications. We, therefore, take these two algorithms as the best algorithms from all that do not use load information, and use them as a baseline for evaluating Feudal Stealing. We show that we can obtain better speedups for irregular parallel applications on heterogeneous computing environments under Feudal Stealing than under Grid-GUM, CRS and Hierarchical Stealing. Additionally, we show that the approximation of the load obtained under Feudal Stealing is better than under Grid-GUM.

Besides the novel Feudal Stealing algorithm, this chapter also makes significant contributions to the study of the state-of-the-art work-stealing algorithms described in Section 2.3.1. As mentioned above, these algorithms were previously evaluated only for very regular (i.e. SimpleDC) divide-and-conquer applications. In this chapter, we evaluate them also for highly irregular parallel applications, which enables us to make new conclusions about their performance. We furthermore, for the first time, evaluate how much their performance could be increased in a hypothetical scenario where PEs have perfect load information at each point in the application execution. This gives a very useful insight for the developers of runtime systems that use these algorithms into how much their runtime systems could benefit if they implement mechanisms for the accurate estimation of PE loads.

5.1 The Use of Load Information

We have already argued in Section 4.4 that, in order to develop an efficient method for locating the steal targets, we need to use information about the dynamic PE loads. In this section, we investigate what is the best way to take advantage of such an information in selecting the steal targets, and how much benefit (in terms of the increased applications' speedups) we can expect from that, compared to the case where no load information is present.

Separating the issue of the use of load information from that of obtaining it enables us to investigate the former under the assumption that *perfect* information is available. By *perfect load information*, we mean that each PE knows exactly what the load of each other PE is, at each point in the application execution. Investigating the applications'

speedups in this settings provides an important sanity check for the issue of obtaining the load information – if, for most of the applications, the speedups that we can obtain with perfect load information are just marginally better than without any load information, then the case for using load information in work-stealing would be lost. Therefore, the questions that we aim to answer in this section are the following:

1. Assuming the presence of perfect load information, what algorithm for selecting the targets should thieves use?
2. How much improvements in speedups we can get with the algorithms that use perfect load information, compared to the algorithms that do not use any load information?
3. What are particularly good or particularly bad cases for the use of load information? In other words, for what applications and on what computing environments do we get the best/the worst improvements in speedups with algorithms that use the perfect load information?

5.1.1 Load-based Work-stealing Algorithms

The question of how to choose a steal target when perfect load information is present is similar to the same question without load information. The only difference is that, with perfect load information, a thief already knows what PEs have work to offload, so it can restrict the set of possible steal targets it considers to only those with work (rather than considering all PEs as possible targets). Then, the question is whether it is the best to choose a *random* target with work, the *nearest* target with work, whether to try stealing from local and remote targets with work *in parallel* in order to do prefetching of work etc. To answer these questions, we consider extensions to the state-of-the-art work-stealing algorithms presented in Section 2.3.1, restricting the set of possible steal targets at each step where a thief needs to choose one to only those that have some work to offload. We also consider two algorithms (CV and HLV) that depend on the presence of load information and that, therefore, do not have an equivalent that does not use load information.

In the following discussion, the algorithms we present are executed by each thief when it needs to select the steal target. Note that we assume that in all algorithms, only one task is sent in each steal operation. The algorithms we consider are:

- **Perfect Random Stealing**. In the basic version of Random Stealing, a thief always chooses a random target from the set of all PEs in a system. In Perfect

Random Stealing, a thief always chooses a random target from the set of all PEs with non-zero load. See Algorithm 9.

Algorithm 9 Perfect Random Stealing

1: S:=set of all PEs that have work to offload
2: Select a random victim v from S
3: Send steal attempt to v

- **Perfect Hierarchical Stealing**. Recall that in Hierarchical Stealing, the set of all PEs in the environment is organised in a PE-tree (based on communication latencies between PEs), and that a thief first looks for work in its whole PE-subtree, and only if no work is found there will it ask its parent for work. Whereas this basic version always looks for work in the whole PE-subtree of the thief before going up in the tree, in the perfect version the thief will only ask its child for work if the child's PE-subtree has non-zero load (that is, if any PE in this PE-subtree has non-zero load). This child PE will then recursively repeat the same procedure (i.e. forward the steal attempt to one of its children whose PE-subtree has non-zero load) until the PE with work is reached. If the whole thief's PE-subtree has zero load, the thief will forward the steal attempt to its parent. In this way, the thief avoids searching for work in its PE-subtrees that do not have any work to offload. See Algorithm 10.

Algorithm 10 Perfect Hierarchical Work Stealing

1: S:=set of all thief's children that have work to offload
2: **if** S is nonempty **then**
3: $\quad v$=random PE from S
4: **else if** thief has parent **then**
5: $\quad v$=thief's parent
6: **else**
7: $\quad v$=NULL
8: **end if**
9: **if** v!=NULL **then**
10: $\quad$ send steal attempt to v
11: **end if**

- **Perfect Cluster-aware Random Stealing**. In Cluster-Aware Random Stealing (CRS), local and remote stealing is done in parallel. That is, a thief looks for work within its own cluster in parallel with looking from work remotely. Similarly to Perfect Random Stealing, in the version of CRS algorithm that uses the perfect load information a thief will always send the steal attempt to a PE

which has some work to offload (if such a PE exists). If the steal attempt needs to be sent within the cluster, a random PE with work from the thief's cluster is selected. Otherwise, in the case of remote stealing, a random PE with work that is outside of the cluster is selected (see Algorithm 11).

Algorithm 11 Perfect Cluster-Aware Work Stealing

1: **if** *outsideStealing* **then**
2: S_1=set of all PEs outside of thief's cluster that have work to offload
3: **if** S_1 is nonempty **then**
4: v_1=random PE from S_1
5: **else**
6: v_1=random PE outside of the thief's cluster
7: **end if**
8: Send steal attempt to v_1
9: outsideStealing = **true**
10: **end if**
11: S_2=set of all PEs from thief's cluster that have work to offload
12: **if** S_2 is nonempty **then**
13: v_2=random PE from S_2
14: **else**
15: v_2=random PE from thief's cluster
16: **end if**
17: Send steal attempt to v_2

- **Perfect Adaptive Cluster-aware Random Stealing**. Recall that in Adaptive Cluster-Aware Random Stealing (ACRS), when doing a remote stealing, the probability that a thief chooses a target from some cluster is proportional to the communication latency between the thief and that cluster (with respect to the latencies to other clusters). In other words, the probability of thief p choosing a target from cluster Q for remote stealing attempt is equal to $\frac{lat(p,Q)}{\sum_{C \in S} lat(p,C)}$, where S is the set of all remote clusters. After the cluster is chosen, the steal attempt is sent to a random PE from it. Therefore, thieves prefer remote stealing from nearer clusters. In Perfect ACRS, the set of all considered clusters is restricted to those that have some PEs with work, and, once the cluster is selected, the steal attempt is sent to a random PE with work from it. See Algorithm 12.

- **Closest-Victim (CV) Stealing**. In this algorithm, a thief always chooses the closest target with work. Note that this algorithm is similar to the perfect version of Grid-GUM work stealing, with the difference that only one task is transferred between thief and victim. In Grid-GUM, a thief chooses the closest target for which it *assumes* (based on the load information it has, which might

be inaccurate or outdated) that it has some work to offload. In CV stealing, the load information that thieves have is perfect, so the closest victim that really has some work is chosen. See Algorithm 13.

- **Highest-Loaded-Victim (HLV) Stealing**. In this algorithm, a thief chooses a target with the highest load (in our case, the largest number of tasks in its task pool). See Algorithm 14.

Note that in all of the algorithms that we consider, it can happen that a thief needs to choose a target, but all of the targets being considered have zero load. For example, it can happen that a thief needs to do local stealing in the CRS algorithm, but all local targets have zero load. In this case, we have decided for Perfect CRS and Perfect

Algorithm 12 Perfect Adaptive Cluster-Aware Work Stealing

1: **if** $outsideStealing$ **then**
2: S_1=set of all remote clusters that have at least one PE with non-zero load
3: **if** S_1 is nonempty **then**
4: $SumLat = \sum_{C \in S_1} \frac{1}{lat(thief,C)}$
5: **for all** $C \in S_1$ **do**
6: $prob(C) = \frac{\frac{1}{lat(thief,C)}}{SumLat}$
7: **end for**
8: D=random cluster from S_1 (where cluster C is selected with probability $prob(C)$
9: v_1=random PE with work from D
10: **else**
11: v_1=random PE outside of the thief's cluster
12: **end if**
13: Send steal attempt to v_1
14: **end if**
15: S_2=set of all PEs from thief's cluster that have work to offload
16: **if** S_2 is nonempty **then**
17: v_2=random PE from S_2
18: **else**
19: v_2=random PE from thief's cluster
20: **end if**
21: Send steal attempt to v_2

Algorithm 13 Closest Victim Work Stealing

1: v=closest PE to the thief that has some work to offload
2: **if** v==NULL **then**
3: v=random PE
4: **end if**
5: Send the steal message to v

ACRS to behave in the same way as their basic versions (i.e. to send the steal attempt to a random PE inside or outside of the cluster), and Perfect Random, CV and HLV stealing to behave in the same way as basic Random Stealing (i.e. to choose the target randomly from the set of *all* possible targets). Other possibilities are to delay sending a steal attempt for some time period, until one of the targets being considered obtains some work, or to send the attempt back to where it originated from and then the thief that had started the stealing might decide to wait for some time and start the stealing process again. We have also tried these possibilities, but did not observe any notable difference in the performance of our algorithms.

Note also that, even with the perfect load information that we assume, it can happen that a target that receives a steal attempt does not have any tasks in its task pool. This can happen if the target's load changes between the time a thief sends the steal attempt to it and the time when the target receives it. In this case, we assume that the target will forward the steal attempt to another appropriately chosen target. In Perfect Random, Perfect Hierarchical, Perfect CV and Perfect HLV, this new target is chosen in the same way as if the original target itself was a thief. In Perfect CRS and Perfect ACRS, the original target will always forward the steal attempt to some PE from its cluster. Additionally, if the steal attempt has visited some predefined number of targets, and it did not find work in any of them, it will be returned to the thief who started it.

Although, of course, the algorithms that we consider here do not cover all possible work-stealing algorithms that use load information, we feel that the scope of considered algorithms is broad enough to make conclusions about the way in which the load information is useful in work-stealing, as they cover most of the state-of-the-art algorithms used on distributed environments.

We will compare the algorithms described above with the basic versions of Random, Hierarchical, ACRS and CRS algorithms[1]. Our specific goals in this section are

[1] In this section we will not consider Grid-GUM, since although it uses the load information, it works with the approximation of PE loads rather than with the perfect information. We will, therefore, postpone considering this algorithm until the next section, where we will compare it with Feudal Stealing.

Algorithm 14 Highest-Loaded-Victim Work Stealing

1: v=PE with the highest load greater than 0
2: **if** v==NULL **then**
3: v=random PE
4: **end if**
5: Send the steal message to v

1. We want to compare the speedups obtained under the Perfect Random, Perfect Hierarchical, Perfect CRS, Perfect ACRS, CV, HLV, Random, Hierarchical, CRS and ACRS algorithms for various applications on various computing environments to discover what algorithm gives the best speedups for most of the application/computing environment combinations. This would give us the answer to what is the best way to choose steal targets if the load information is present during the application execution.

2. Additionally, we want to compare the Perfect Random, Perfect Hierarchical, Perfect CRS and Perfect ACRS algorithms against the Random, Hierarchical, CRS and ACRS algorithms, respectively, in order to find out how much the speedups under each of these algorithms can be improved if the perfect load information is used in them. This would give us the indication of how much better performance can we expect from each individual basic algorithm we consider (Random, Hierarchical, CRS and ACRS) if they have access to information about the load of the environment. This is useful for the developers who must use one of these basic algorithms in their systems, so that they can estimate whether it would be useful to consider extending these algorithms with the mechanisms for estimating the load information during the application execution.

Concerning the methodology of evaluating the performance of algorithms, our aim to explore the use of perfect load information drives this decision towards the use of simulations. In the real runtime systems on distributed computing environments, due to the communication latency, it is generally not possible for PEs to have totally accurate load information of the rest of the environment. Even in the case where each PE sends its load information to some central PE (to which also all steal messages are sent), by the time the message with load information arrives to this central PE, the load of the PE from which the message originates might have changed. This is especially the issue in the systems with high communication latencies. In simulations, however, we can assume that each PE has access to some kind of global system state, where the information about the loads of all PEs is kept. This enables the thieves to have accurate information about the load of all possible targets at the time when they need to decide where to send steal requests.

5.2 Evaluation of Load-based Work-stealing Algorithms

It initially appears that having load information should increase the speedups of all applications under all work-stealing algorithms on all computing environments. After all, if a thief has information about the load of all possible targets, then it knows where to look for work, and it should certainly be able to obtain work faster than if it goes looking for work blindly. Therefore, the thieves should be idle for less time, resulting in their increased utilisation and, therefore, better applications' speedups. However, many cases exist where this is not the case. For example, in the evaluation of Grid-GUM work-stealing in [AZ06], it is concluded that in homogeneous computing environments with small latencies between PEs, having load information does not bring much benefit to the performance of Random Stealing. Although, with load information, many fewer steal messages are exchanged, the cost of sending individual messages is very small (due to the low latency), so reducing the number of messages sent does not have much impact on the speedups of parallel applications. We will also see that even on heterogeneous environments, in the cases of applications where parallel tasks generate a lot of additional parallelism (for example, in simple divide-and-conquer applications), the use of load information also does not bring much benefit to most of the algorithms we consider. In this case, since most of the PEs have a lot of parallel tasks in their queues, randomly choosing a steal target is very likely to hit the one with work. Therefore, it is not trivial to ask what applications under what work-stealing algorithms could benefit from the use of load information, and on what computing environments would the benefit be the biggest.

We will focus on the applications conforming to the tree-like model of tasks, described in Section 4.2. Our main assumption is that the use of load information can significantly improve the speedups of applications that generate substantially larger number of sequential than nested-parallel tasks, i.e. which are highly irregular. Since there are fewer nested-parallel tasks, we expect that during the execution of these applications parallel work will be concentrated on fewer PEs. Because of this, it would be harder for algorithms that choose steal targets randomly to locate the ones with work. Hierarchical Stealing would have even bigger problems, since there is high probability that the whole PE-subtrees of many thieves would be idle. Therefore, we expect that use of load information would enable PEs to spend much less time looking for work (and, potentially, send many fewer messages over high-latency networks), hence improving the speedups of applications. Our focus will, therefore, be on the SimpleDC

and DCFixedPar applications.

We will focus mostly on applications with finer-grained sequential tasks (with respect to wide-area latencies in the environment), as many more steal attempts need to be made during the execution of these applications (due to a short time needed to execute each sequential task) than for the ones with coarse grained tasks.

Regarding the computing environments simulated, we made two hypotheses:

1. We expected that the better improvements with load information will be obtained on the environments consisting of larger number of PEs, since there is more chance of non-load-based stealing going "wrong" (i.e. choosing the wrong steal targets) in this setup. To verify the hypothesis, we have considered environments consisting of varying number of clusters and varying number of PEs in each cluster. We will denote by Grid(X,Y,LANLat,WANLat) environments which consists of X different clusters, each of which consists of Y PEs, and where the latency between every two PEs that belong to the same cluster is LANLat, and the latency between every two PES belonging to different cluster is WANLat. In most of the experiments in this section, we will focus on the systems where WANLat was set to $10ms$, and the LANLat to $0.1ms$. Therefore, Grid(X,Y) will denote the environment where latency between every two clusters is $10ms$ and the latency inside all clusters is $0.1ms$. We have chosen these latencies to represent the environments where communication over WAN is significantly (100 times) higher than over LAN. This choice of latencies models the environment where different clusters from one country are linked together into a large Grid[2].

2. For the environments consisting of the same number of PEs grouped in the same number of clusters, we expected to see better improvements on environments with higher and more-heterogeneous latencies. To verify this, we have considered distributed environments consisting of 8 clusters, with 8 PEs in each cluster and with various groupings of clusters based on the latency between them. See Figure 5.1 for the environments of this kind that we have considered.

We will group the results obtained with experiments into two subsections. Subsection 5.2.1 focuses on the applications which generate large amounts of parallelism. In this subsection, we will consider the SimpleDC applications. Results for the SimpleDC applications are further divided according to the computing environments simulated. The first group of results evaluates the work-stealing algorithms we study on the envi-

[2] 10ms was an average latency obtained after testing the latency between servers at several institutions in the UK

Comp. environment	**Description**
WorldGrid-Hom	Homogeneous system, latency between every two PEs is 0.1ms
WorldGrid-Uni-10ms	Latency between every two clusters is 10ms (Same as Grid(8,8))
WorldGrid-2L-20ms	Clusters split into two continental groups of 4 clusters each, latency between them being 20ms. Within each continental group, latency between every two clusters is 10ms
WorldGrid-2L-30ms	Same as WorldGrid-2Levels-20ms, but with latency between continental groups being 30ms
WorldGrid-2L-50ms	Same as WorldGrid-2Levels-20ms, but with latency between continental groups being 50ms
WorldGrid-3L-80ms-30ms	Clusters split into two continental groups of 4 cluster each, latency between them being 80ms. Each continental group is split into two country groups (with 2 clusters in each group), with the latency between them being 30ms. Each country group split into two site-clusters, latency between them being 10ms (See Figure 5.2)

Figure 5.1: The WorldGrid computing environments simulated. Each environment consists of 8 clusters, each of which consists of 8 PEs

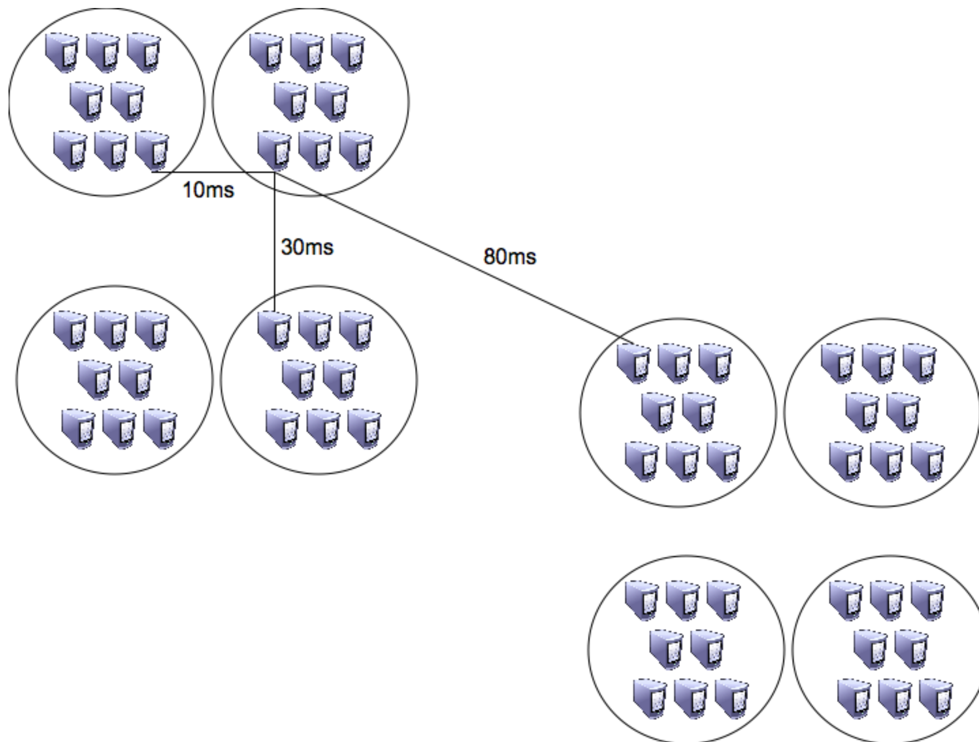


Figure 5.2: WorldGrid-3L-80ms-30ms

ronments with the fixed number of PEs per cluster, fixed communication latency between clusters and variable number of clusters (Grid(1,8)–Grid(8,8)), the second group studies the environments with fixed number of clusters, fixed communication latency between clusters and variable number of PEs in each cluster (Grid(8,4), Grid(8,6), Grid(8,8), Grid(8,10), Grid(8,12), Grid(8,14) and Grid(8,16)), and the third group studies the environments with fixed number of cluster, fixed number of PEs in each cluster and variable latency between clusters (the WorldGrid environments). Subsection 5.2.2 focuses on more irregular applications, where there is not as much parallelism as in the SimpleDC applications. In this section, our focus is on the DCFixedPar applications. Furthermore, since we have an additional parameter for these applications (which is the degree of their regularity), in order to make the number of experiments tractable, we restrict our attention to large computing environments with heterogeneous latencies. For this reason, in the experiments with the DCFixedPar applications, we only consider the WorldGrid computing environments.

Note that, due to a degree of randomness present in all work-stealing algorithms we consider (except for Hierarchical Stealing), it is possible even under simulations to get different speedup for the same application on the same computing environment under the same work-stealing algorithm. In each experiment we have, therefore, taken

the average speedup over 10 executions of the same application on the same environment under the same work-stealing algorithm, except where we evaluated Hierarchical Stealing, where only one execution was needed, since this algorithm is deterministic.

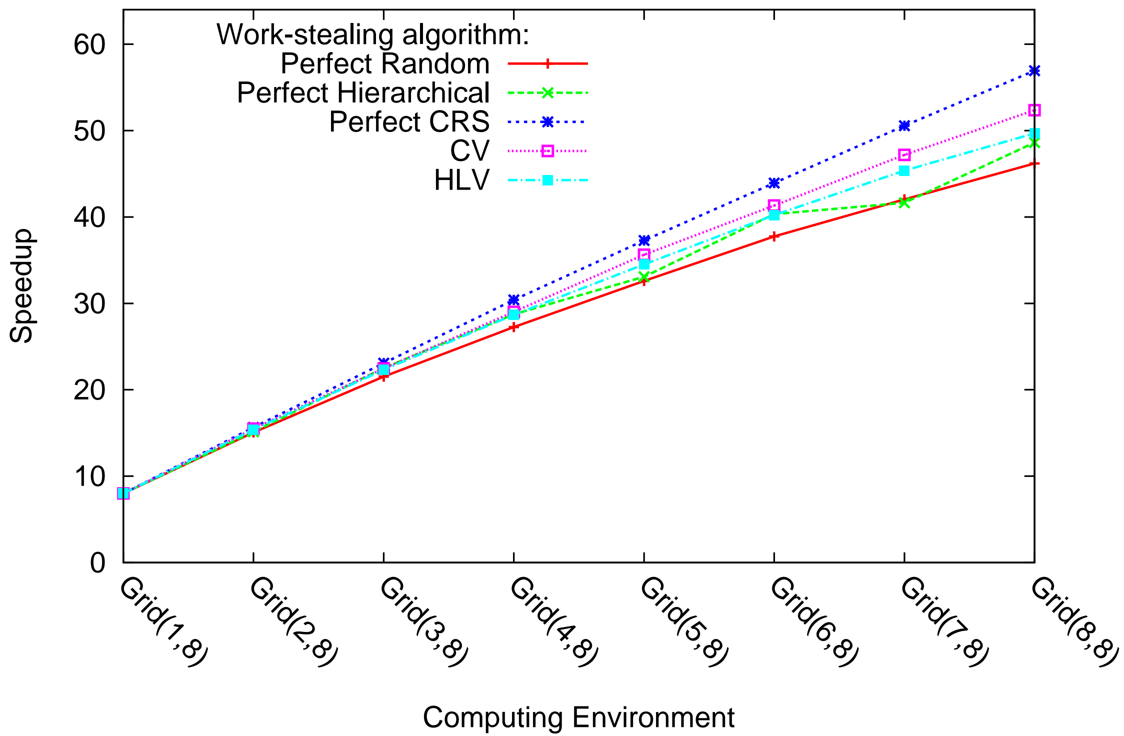
5.2.1 SimpleDC Applications

In our first experiment, we consider the SimpleDC(12,5ms) application. This is the example of an application whose all tasks (except for the ones generated after the threshold level in task tree is reached) are nested-parallel. Consequently, most of the application's tasks are coarse grained and they create a lot of additional parallelism. This further means that most of the thieves that manage to steal tasks will create additional parallel tasks, making them potential victims for other thieves. We, therefore, expect the number of potential victims to be large for most of the application execution. Additionally, due to a coarse granularity of tasks, thieves that obtain work are kept busy for long time. Consequences of all this are that we expect that PEs will not have to steal very often, and even when they need to, random stealing techniques that do not use load information will be successful in locating work. This application, therefore, represent somewhat the worst case for the algorithms that use perfect load information, since we did not expect to see notable improvements in speedups under them.

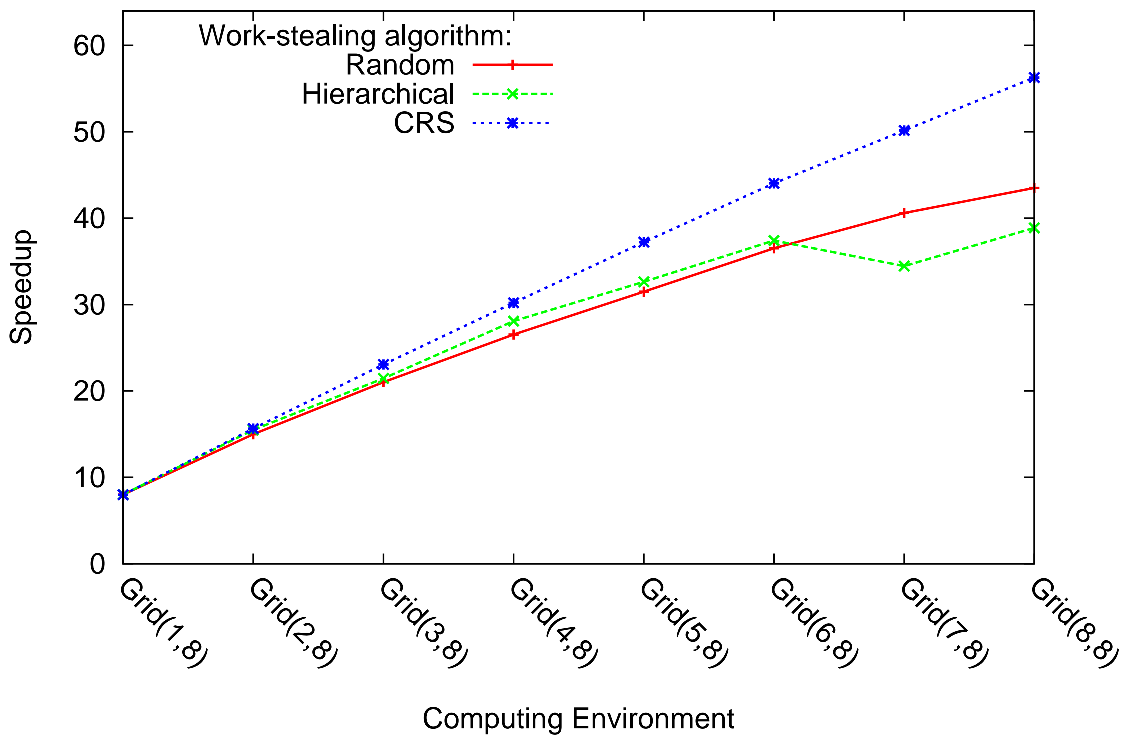
Grid(1,8)–Grid(8,8) environments

Figure 5.3 shows the speedups obtained under the algorithms both with (Figure 5.3a) and without (5.3b) perfect load information on the Grid(1,8)-Grid(8,8) environments. Note that we have omitted the Perfect ACRS algorithm from Figure 5.3a (and also the ACRS algorithm from the Figure 5.3b) since, because of the uniform latency between all clusters, on these environments they are identical to Perfect CRS and CRS. From the Figure 5.3a, we can see that for smaller number of clusters, all algorithms with perfect load information perform similarly, but as the number of clusters increases, Perfect CRS starts to outperform the other algorithms. This is a perfect example of the benefit of doing local and remote stealing in parallel, since most of the times thieves are able to obtain work locally, and the latency in prefetching work remotely is successfully hidden by executing the locally obtained work. We can also see that all algorithms scale well as the number of clusters in an environment increases.

From Figure 5.3b, we can see that, similarly to when perfect load information is used, when no load information is used the CRS algorithm gives the best speedup, and



(a) Perfect Load Information



(b) No Load Information

Figure 5.3: The SimpleDC(12,5ms) application on the Grid(n,8) environments

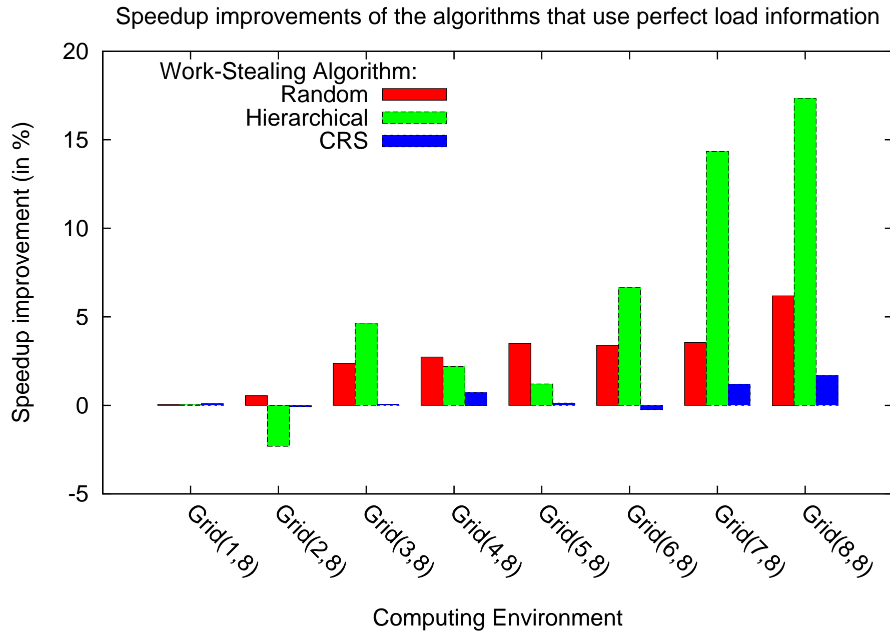


Figure 5.4: Speedup improvements of the algorithms that use perfect load information over those that do not use load information for the SimpleDC(12,5ms) application on the Grid(n,8) environments

the difference between it and other algorithms becomes bigger when more clusters are added to the environment. An interesting point that we can observe from the figure is that Hierarchical Work Stealing fails to scale to larger environments (in this case, Grid(7,8) and Grid(8,8)), and we can even see that the speedups under it are worse on the Grid(7,8) than on the Grid(6,8) environment.

From the two considered figures, we can conclude that for simple divide-and-conquer applications, the CRS algorithm gives the best speedups both in the presence of perfect load information and without it on computing environments where there is a uniform latency between clusters.

Figure 5.4 shows the speedup improvements that the use of load information brings to the SimpleDC(12,5ms) application under the Random, Hierarchical and CRS algorithms for the computing environments considered on Figures 5.3a and 5.3b. In other words, it shows the difference in speedups under Perfect Random and Random, Perfect Hierarchical and Hierarchical, and Perfect CRS and CRS algorithms. We can observe the following facts for the three considered algorithms:

- For the CRS algorithm, the use of load information does not bring almost any benefit, and in the case of Grid(6,8) it even brings a marginal decrease in perfor-

mance. The reason for this is not clear, as the difference in speedup is too small for us to be able to investigate the reasons for it. In any case, the CRS algorithm without any load information performs very well for the SimpleDC(12,5ms) application, giving close-to-linear speedups on all of the environments considered at the Figure 5.3b, so there is not much space for the improvements when load information is added to it.

- For Random Stealing, the use of load information brings small but measurable improvement on the computing environments comprising a larger number of clusters (Grid(3,8)–Grid(8,8)), whereas for the environments comprising a smaller number of clusters, the improvements are minimal. This shows that, although Random Stealing performs very well for the SimpleDC applications, on larger systems it might be worthwhile using the load information, since random selection of targets can introduce a small decrease in performance.

- For Hierarchical Stealing, we can see notable improvements when load information is used on the environments comprising a larger number of clusters. We can see that the improvements on Grid(7,8) and Grid(8,8) are 15% and 17%, respectively.

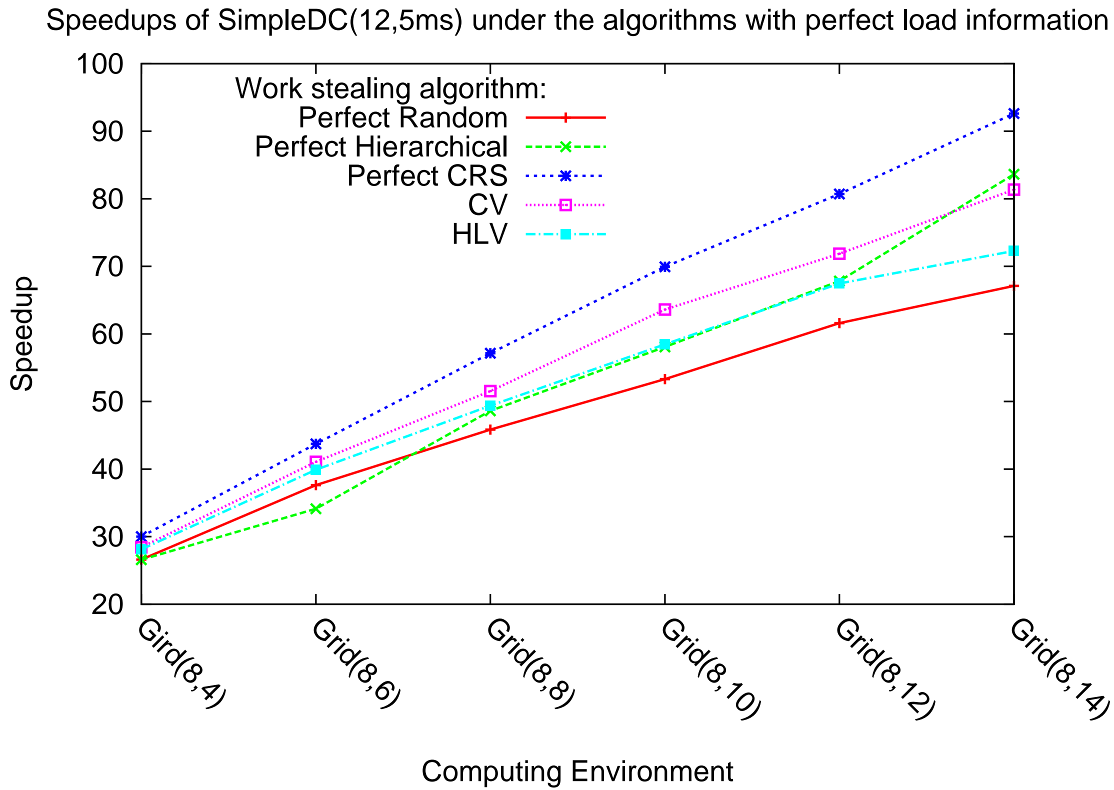
From this experiment we can, therefore, conclude the following facts for the applications whose all tasks generate a lot of parallelism (except, of course, the sequential ones at the end of execution), on the computing environments with homogeneous latencies between clusters:

1. The CRS algorithm gives the best speedups.

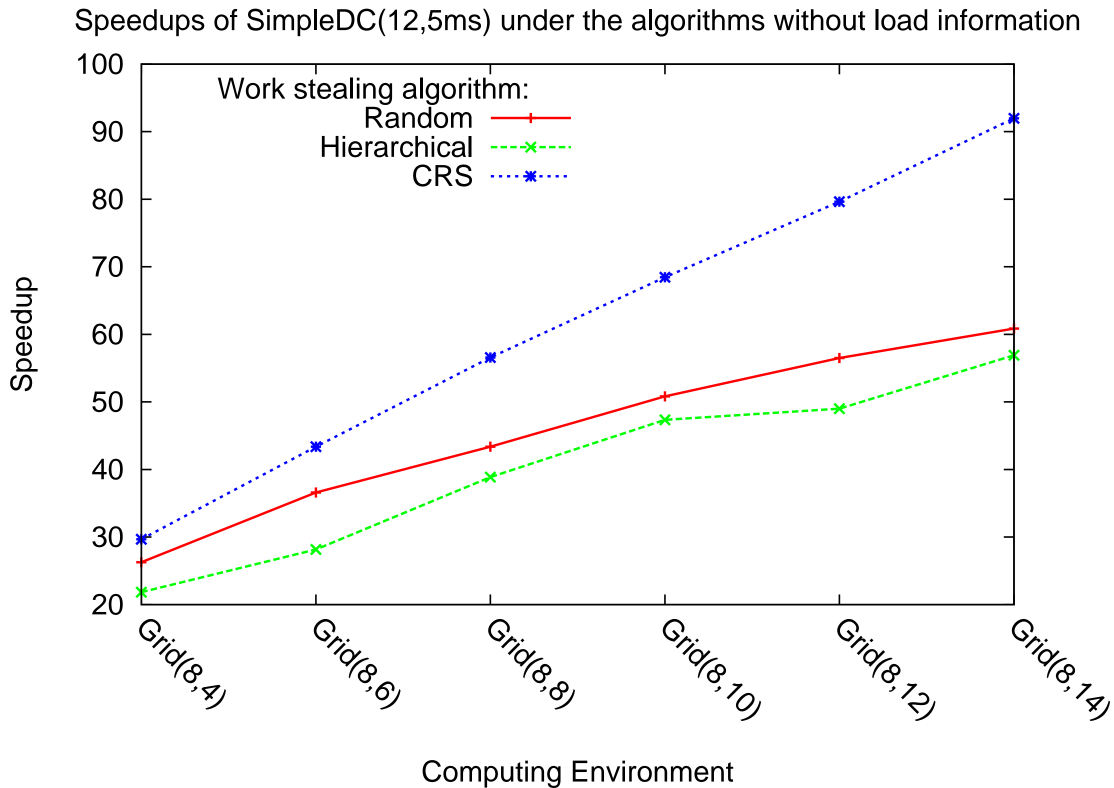
2. The algorithms that use some form of random stealing, which do not use load information, give a very good load distribution, and the applications' speedups cannot be notably improved when the load information is used.

3. Hierarchical Stealing can notably benefit from the use of load information, as without it the algorithm fails to scale for the environments with larger number of clusters.

Our conclusions for the algorithms that do not use load information generally agree with those in Van Nieuwpoort et al. [VNKB01].



(a) Perfect Load Information



(b) No Load Information

Figure 5.5: Speedups of SimpleDC(12,5ms) on Grid(8,n) environments

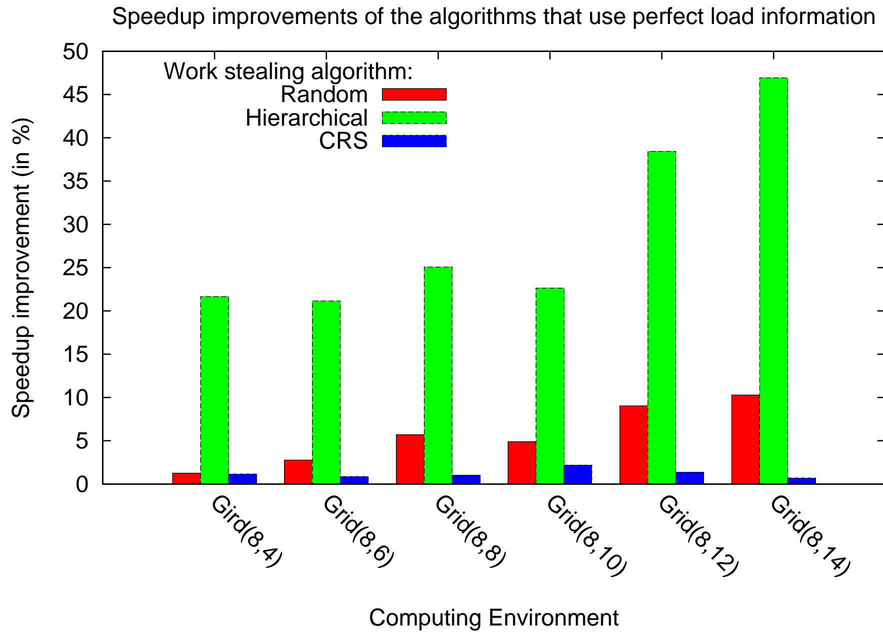


Figure 5.6: Speedup improvements of the algorithms that use perfect load information over those that do not use load information for the SimpleDC(12,5ms) application on the Grid(8,n) environments

Grid (8,n) environments

Figure 5.5 shows the speedups of the SimpleDC(12,5) application on the Grid(8,n) environments, for $n \in \{2, 4, 6, 8, 10, 12, 14\}$. This figure shows us how do the work stealing algorithms scale on the environments that have an increasing number of PEs in each cluster. We can see that, when perfect load information is used (Figure 5.5a), all algorithms scale reasonably well on these environments. However, we can again see (as for the Grid(n,8) environments) that the CRS algorithm outperforms all others, and the difference between it and the other algorithms becomes bigger the more PEs there are in the clusters. This confirms what was indicated in the previous experiment, which is that CRS scales the best to larger environments for applications that generate a lot of parallelism.

We can observe similar situation for algorithms without load information (Figure 5.5b). Again, the CRS algorithm scales the best as the number of PEs in clusters increases, and the difference between it and the Random and Hierarchical Stealing gets higher the more PEs there are in clusters.

One thing that we can observe from Figure 5.5b is a particularly bad performance of Hierarchical Stealing, which is most of the time worse than even Random Stealing.

Furthermore, Hierarchical Stealing has problems with scaling on some environments, as we can see, for example, that it gives the same speedup on the Grid(8,10) as on the Grid(8,12) environment. This is due to a slower distribution of work under Hierarchical Stealing on larger environments. On larger environments, PE-trees for most of the PEs are also larger and, consequently, it takes more time for each thief to traverse his whole PE-tree before going up in the tree. As a result of this, at the beginning of the application execution it takes more time for most of the thieves to reach the cluster where the main task starts execution and where, consequently, all of the parallelism is. These thieves will, therefore, acquire work more slowly and, more importantly, they will steal tasks that generate less parallelism (since tasks that generate larger amounts of parallelism will already be stolen by the PEs from the main cluster). This is the reason for the poor scalability of Hierarchical Stealing on most of the environments with larger number of PEs per cluster.

Overall, from this experiment we can get to the same conclusions as in the previous one: The CRS algorithm gives the best speedups and has the best scalability of all considered algorithms, both with and without load information.

Figure 5.6 shows the improvements in speedups that the use of perfect load information brings to the Random Stealing, Hierarchical Stealing and CRS algorithms. For the CRS algorithm, as in the previous experiment, we can see that the improvements are marginal, being between 1% and 3%. For Random Stealing, we can observe more notable improvements, especially on the environments with bigger number of PEs per cluster, where the improvements are above 10%. This is somewhat expected, as the more PEs are there in the environment, the more chance there is for random stealing to go wrong, so the use of perfect load information can indeed help on these environments. For Hierarchical Stealing, we can observe very good improvements (up to 30%) when load information is used, and that the improvements get higher as the number of PEs per cluster increases. The previously mentioned problems of thieves spending too much time traversing their PE-subtrees are avoided when the load information is used, and therefore the improvements that Perfect Hierarchical Stealing brings to the basic Hierarchical Stealing are very good on larger environments.

From the Figures 5.5 and 5.6, we can make similar conclusions for the applications that create a lot of parallelism on the environments with different number of PEs per cluster as for the environments with fixed number of PEs per clusters and different number of cluster:

- On both kinds of environments, the CRS algorithm gives the best speedups both with and without load information, and it cannot be notably improved with the

use of perfect load information.

- For Random Stealing, on the environments with smaller number of PEs per cluster, we get only marginal improvements. However, on the environments with bigger number of PEs per cluster, we get notable improvements (above 10%) when perfect load information is used.

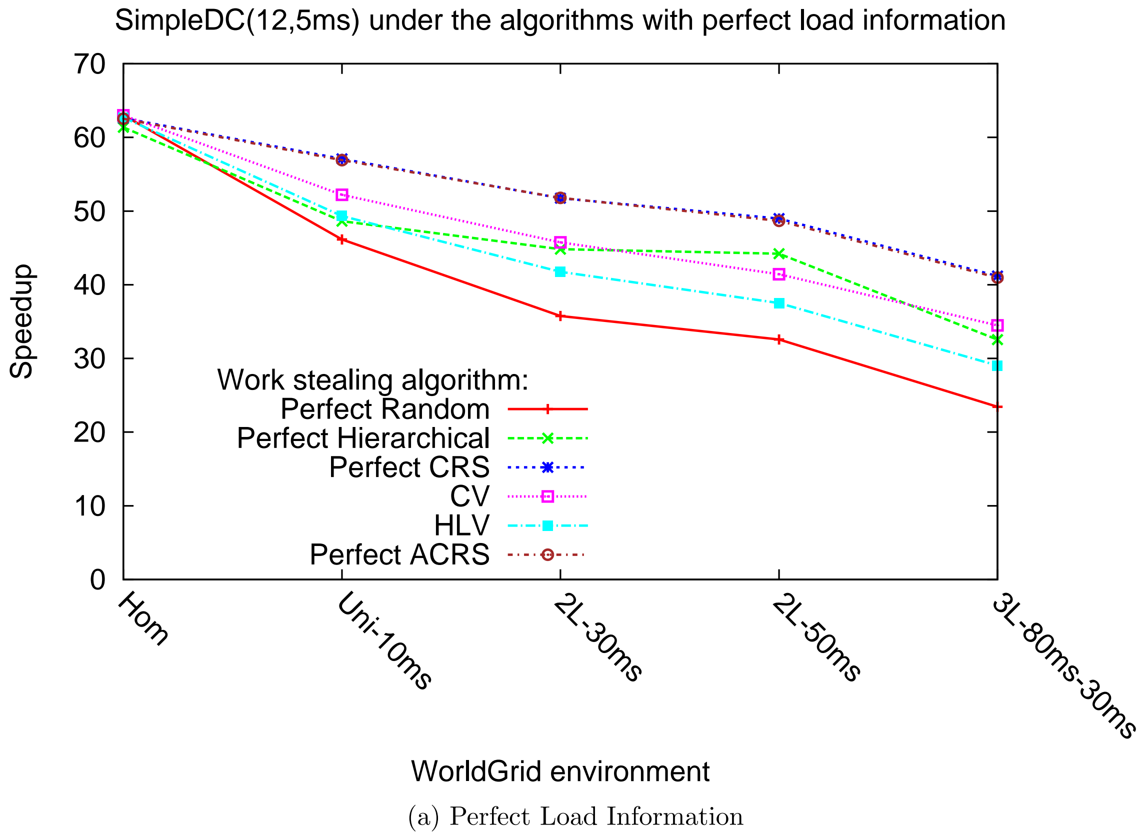
- For Hierarchical Stealing, the use of load information seems very important, as it brings significant improvements to the speedups, especially on the environments with bigger number of PEs per clusters, where we have observed the improvements of 20-30%.

The WorldGrid environments

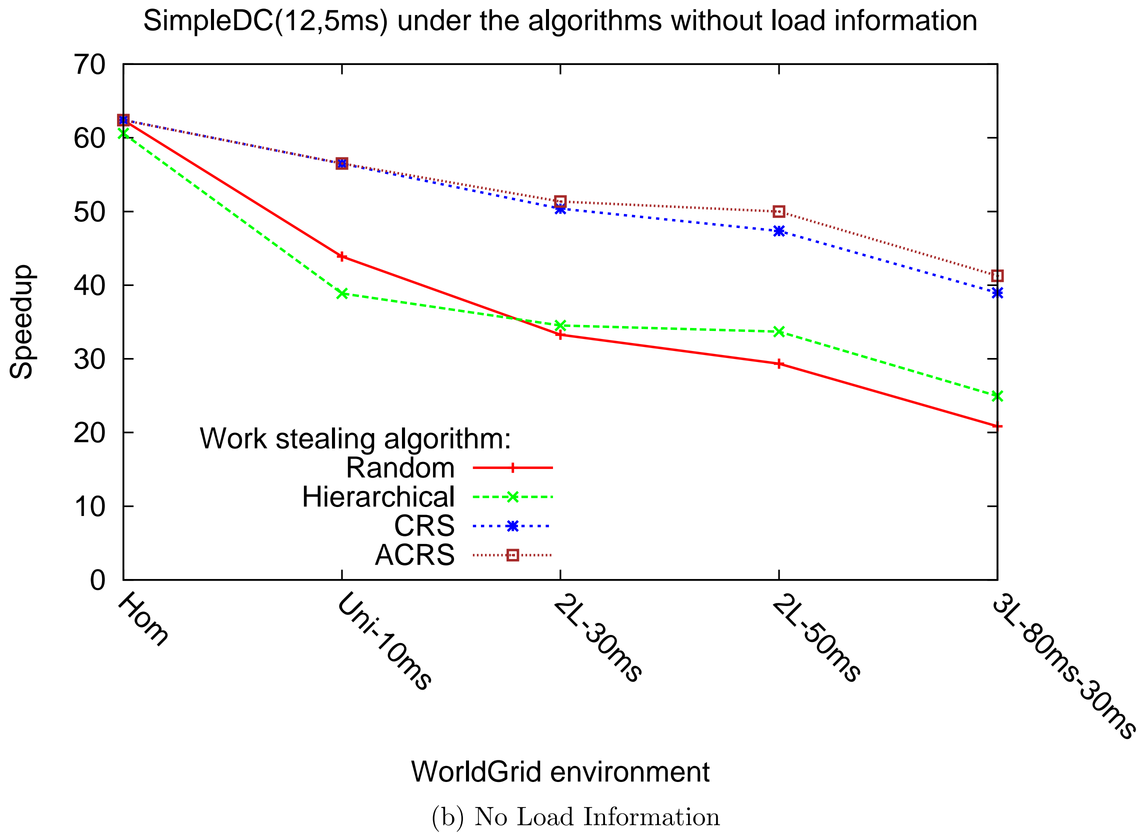
Figure 5.7 shows the speedups of the SimpleDC(12,5ms) application on the WorldGrid environments from Figure 5.1 (page 103). Again, as in the previous experiments, Figure 5.7a shows the speedups under the algorithms that use perfect load information, and 5.7b under the ones that do not use load information. Compared to the previous two experiments, here all of the environments have the same number of PEs, but different environments have different number of communication latency levels (from 1 level on WorldGrid-Hom, where the latency between all PEs is the same, to 4 levels on WorldGrid-3L-80ms-30ms). Consequently, since the latencies between the clusters are not uniform any more, in this experiment we also include the ACRS and Perfect ACRS algorithms, since on these environments they will differ from CRS and Perfect CRS.

From Figure 5.7, as we have expected, we can observe that the speedups under all algorithms are dropping as the computing environment gets more heterogeneous. We can again observe that the CRS algorithms gives the best performance both with and without load information. We can also observe that, without load information, ACRS performs slightly better than CRS on more heterogeneous environments. With perfect information, however, CRS works slightly better. The differences are, however, minor. Very similar performance of the ACRS and CRS algorithms can be explained by the fact that most of the times PEs are able to obtain work locally, which means that the latency from wide-area steal attempts is successfully hidden by executing that work. Therefore, it does not make much difference whether thieves prefetch work (via wide-area stealing) from closer or further clusters.

Similarly to Figure 5.4, Figure 5.8 shows the improvements that the use of perfect load information brings to the speedups of the SimpleDC(12,5ms) application on the



(a) Perfect Load Information



(b) No Load Information

Figure 5.7: The SimpleDC(12,5ms) application on the WorldGrid environments

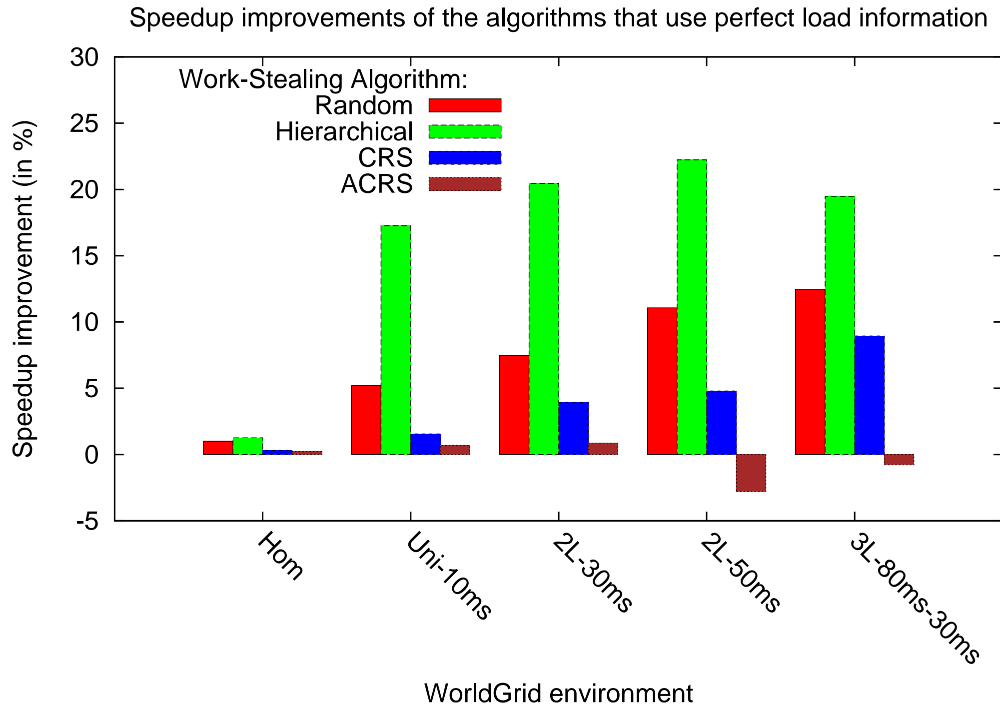


Figure 5.8: Speedup improvements of the algorithms that use perfect load information over those that do not use load information for the SimpleDC(12,5ms) application on the Grid(8,n) environments

WorldGrid environments for the considered algorithms. We can again observe that the improvements to the CRS (and, this time, also for the ACRS) algorithm are only marginal, except on the WorldGrid-3L-80ms-30ms environment, where the improvement of Perfect CRS over CRS is 9%. The improvements for Random Stealing and Hierarchical Stealing are much more significant, being up to 15% and 25% respectively. Not surprisingly, for Random Stealing the improvements are higher the more heterogeneous computing environment is, since in the presence of higher latencies, bad random selection of steal targets has worse impact on the application speedup. For Hierarchical Stealing, however, there does not seem to be a clear correlation between the amount of improvement obtained when the load information is used and the heterogeneity of the environment. We can clearly see that the improvements on heterogeneous environments are much higher than in the case of the WorldGrid-Hom environment. However, the improvements on all heterogeneous environments are about the same. This can be explained by the fact that hierarchical organisation of PEs into PE-trees and systematic way of searching these trees for work (from closer to more farther PEs) in Hierarchical Stealing successfully hides the heterogeneity in communication latencies,

and, therefore, speedups under Hierarchical and Perfect Hierarchical Stealing are very similar on all of the WorldGrid environments (except, of course, for WorldGrid-Hom). As a consequence of this, the improvements of Perfect Hierarchical over Hierarchical Stealing are very similar on all environments.

From the experiments with the WorldGrid environments, we can make the following conclusions:

- The Perfect CRS algorithm gives the best speedups. On most of the environments, the speedups under the CRS and Perfect CRS algorithms are very similar. However, on highly heterogeneous environments, Perfect CRS notably outperforms CRS, which means that on these environments the CRS algorithm can be notably improved with the use of perfect load information.

- Random and Hierarchical Stealing, on the other hand, can be significantly improved with the use of perfect load information. For Random Stealing, the improvements with perfect load information are better the more heterogeneous the environment is, whereas for Hierarchical Stealing, this is not the case. The improvements there are uniform for all the heterogeneous computing environments we have considered.

Other SimpleDC Applications

The SimpleDC(12,5ms) application is an example of the application that generates a lot of parallel tasks (since the task tree has 13 levels, the total number of tasks in this application is 8192) and where there are many nested-parallel tasks (half of the tasks, or 4096 in total, are nested-parallel). Also, as we have explained before, this application is somewhat an ideal case for the use of Cluster-Aware Random Stealing, since, because of a lot of nested-parallel tasks, there are a lot of potential victims in the environment and the thieves are usually able to obtain work locally, so prefetching the work over high-latency networks, which is done in parallel with local stealing, comes almost for free. This is the reason we did not observe any notable improvements in speedups when the CRS and ACRS algorithms use perfect load information.

It is interesting to observe what happens for the applications of the same type, but which generate fewer parallel tasks. In the case of the SimpleDC(T,C_{seq}) applications, this corresponds to lower level-threshold value T. Since there is less parallelism available, obtaining the work locally, while at the same time another work from remote targets, might not be as easy as for applications with more parallelism. Additionally, we also want to investigate what difference does the use of load information make

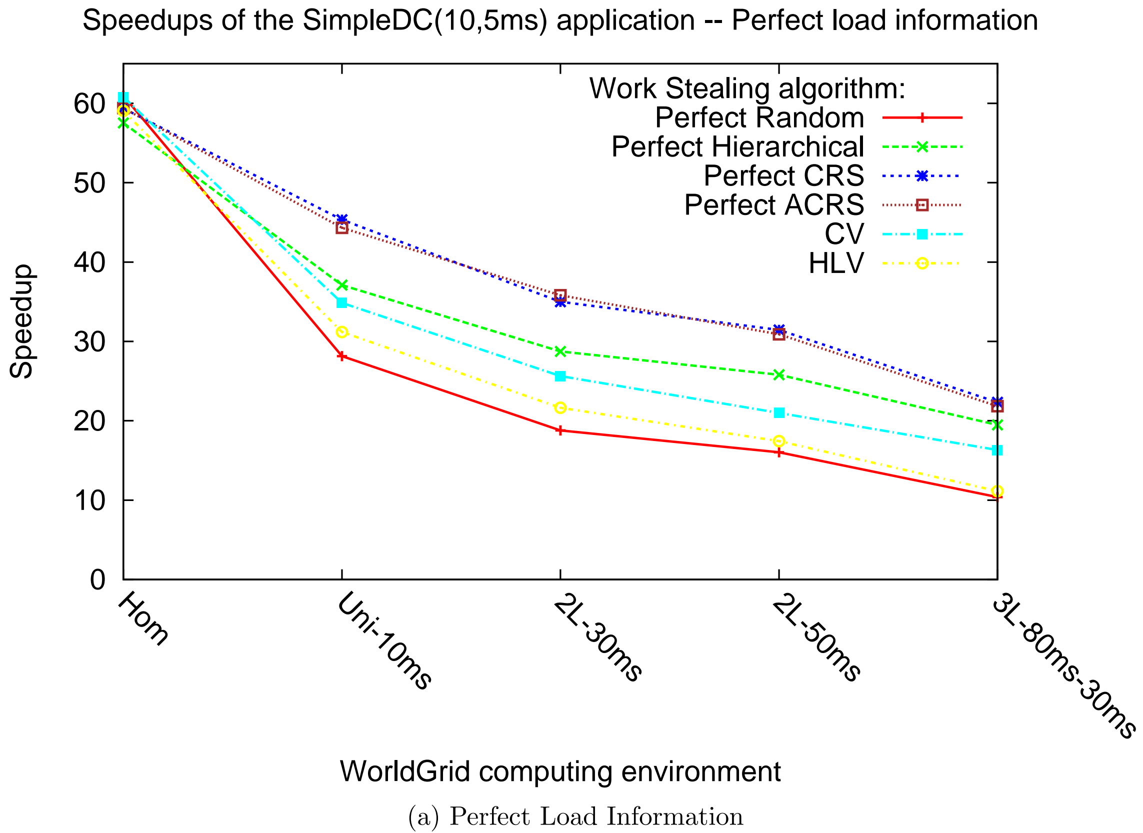
on the speedups of SimpleDC applications that have coarse-grained sequential tasks. For these reasons, we have considered the SimpleDC(10,5ms) and SimpleDC(10,30ms) applications. These applications have fewer number of tasks than SimpleDC(12,C_{seq}) (2048 compared to 8192) and, consequently, fewer nested-parallel tasks (1024 compared to 4096).

Figure 5.9 shows the speedups of the SimpleDC(10,5ms) application on the WorldGrid computing environments under the algorithms both with (Figure 5.9a) and without (Figure 5.9b) load information. The first thing that we can notice when comparing the speedups of SimpleDC(10,5ms) with speedups of SimpleDC(12,5ms) (Figure 5.7 on page 113) is that the speedups of SimpleDC(10,5ms) are lower under all algorithms on considered computing environments. This is what we expected, since there are fewer nested-parallel tasks in the SimpleDC(10,5ms) application, and, therefore, thieves end up stealing more sequential tasks from victims, which means that they need to steal more often and are idle for more time.

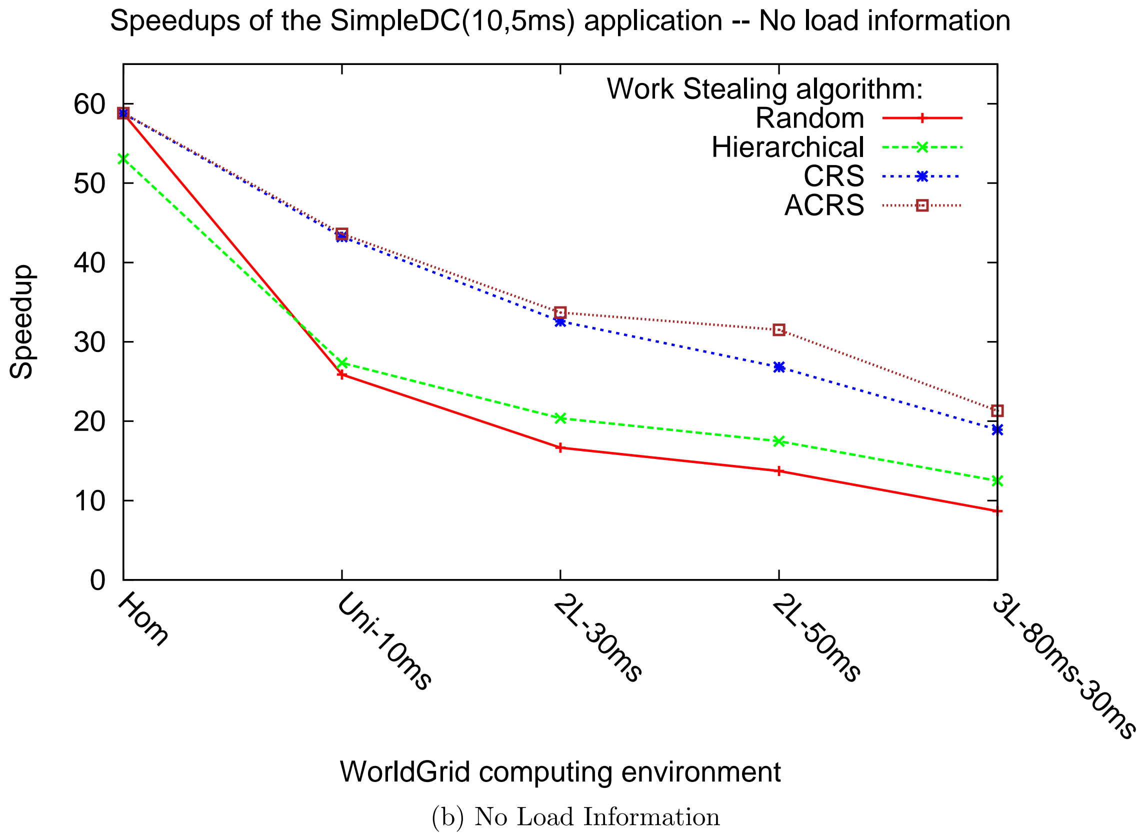
Concerning the performance of individual algorithms, we can see an identical situation on Figures 5.9 and 5.7. The Perfect CRS and Perfect ACRS algorithms give very similar speedups, and they perform notably better than the other algorithms. Without load information, on more heterogeneous environments, ACRS outperforms all other algorithms. On more homogeneous environments, the performance of CRS and ACRS algorithms is very similar.

Figure 5.10 shows the improvements in speedups that the perfect versions of algorithms bring over the non-perfect ones for the SimpleDC(10,5ms) application on the WorldGrid environments. We can see that the improvements for all algorithms (except for ACRS) are better than in the case of the SimpleDC(12,5ms) application. The improvements for Hierarchical Stealing are up to 55% on the WorldGrid-3L-80ms-30ms environment, and we can also observe the notable improvements for the CRS algorithm (up to 20%, again on the WorldGrid-3L-80ms-30ms environment).

Reasons for better improvements of perfect algorithms for the SimpleDC(10,5ms) application lie in bad performance of most of the non-perfect algorithms for this application. Specifically, since there are less nested-parallel tasks in it that in SimpleDC(12,5ms), and since the thieves from the cluster where the main PE is are quicker to obtain work when the application execution starts, the thieves from other clusters are much more likely to obtain tasks that do not create a lot of additional parallelism. This fact, coupled with the fact that sequential tasks are very fine grained, means that the thieves from clusters other than the one where the main PE will usually steal smaller chunks of work than in the case of the SimpleDC(12,5ms) application,



(a) Perfect Load Information



(b) No Load Information

Figure 5.9: Speedups of the SimpleDC(10,5ms) application on the WorldGrid environments

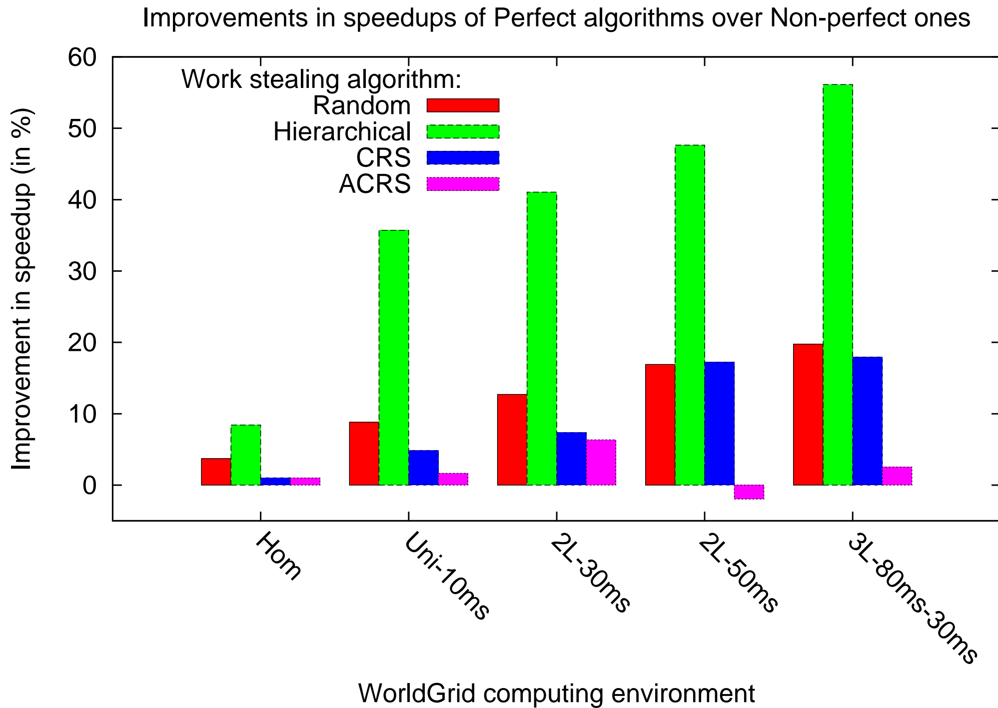


Figure 5.10: Improvements in speedups with perfect load information of the SimpleDC(10,5ms) application on the WorldGrid environments

which means that they will have to look for work more often. Also, since there is less parallelism available, thieves are less likely to hit the target with work than for the SimpleDC(12,5ms) application. The use of load information helps with both of these problems. Since the thieves know exactly where to look for work, they are more likely to obtain tasks that generate more parallelism before they are stolen by thieves from the main cluster. Also, the thieves are much more likely to hit the targets with work if the load information is used. As a consequence of all this, the improvements in speedup with perfect load information are better for the SimpleDC(10,5ms) application.

We can also observe that (similar to Figure 5.8 on page 114, which shows the improvements for the SimpleDC(12,5ms) application) the improvements for all of the algorithms (except for ACRS) are higher the more heterogeneous computing environment is. In contrast to Figure 5.8, in this case we can also observe that the improvements for Hierarchical Stealing are increasing as the computing environment gets more heterogeneous. This is due to a less parallelism being available in the application, and therefore thieves having to search larger portions of their PE-trees when no load information is used. For many thieves, traversing PE-tree is more expensive (in terms of total time it takes to ask all of the targets in it for work) on more heterogeneous

environments with the higher communication latencies. This problem is, of course, avoided if perfect load information is used.

Finally, Figure 5.11 shows the speedups under the considered algorithms for the SimpleDC(10,30ms) application on the WorldGrid computing environments. Compared to SimpleDC(10,5ms), this application has coarser-grained sequential tasks. Therefore, thieves are kept busy for longer time when they steal tasks than in the SimpleDC(10,5ms) application, which means that they do not have to steal as often. Consequently, we can observe much better speedups under all the algorithms than on Figure 5.9. Additionally, if we look at the speedup improvements that the algorithms that use perfect load information bring (Figure 5.12), we can see that the improvements are smaller than in the case of SimpleDC(10,5ms), and are very similar to these for SimpleDC(12,5ms) (see Figure 5.8 on page 114). This is due to a better performance of the algorithms that do not use load information. Although there is less parallelism in SimpleDC(10,30ms) than in the SimpleDC(12,5ms) application, and, therefore, thieves can more easily miss steal victims, sequential tasks are of larger size, so there is no danger of stealing too fine-grained tasks.

From the experiments with the SimpleDC(10,5ms) and SimpleDC(10,30ms) applications, we can make the following conclusions about the performance of considered algorithms for applications which all tasks are nested-parallel, but which do not have as much parallelism as the larger ones, on the environments with heterogeneous communication latencies:

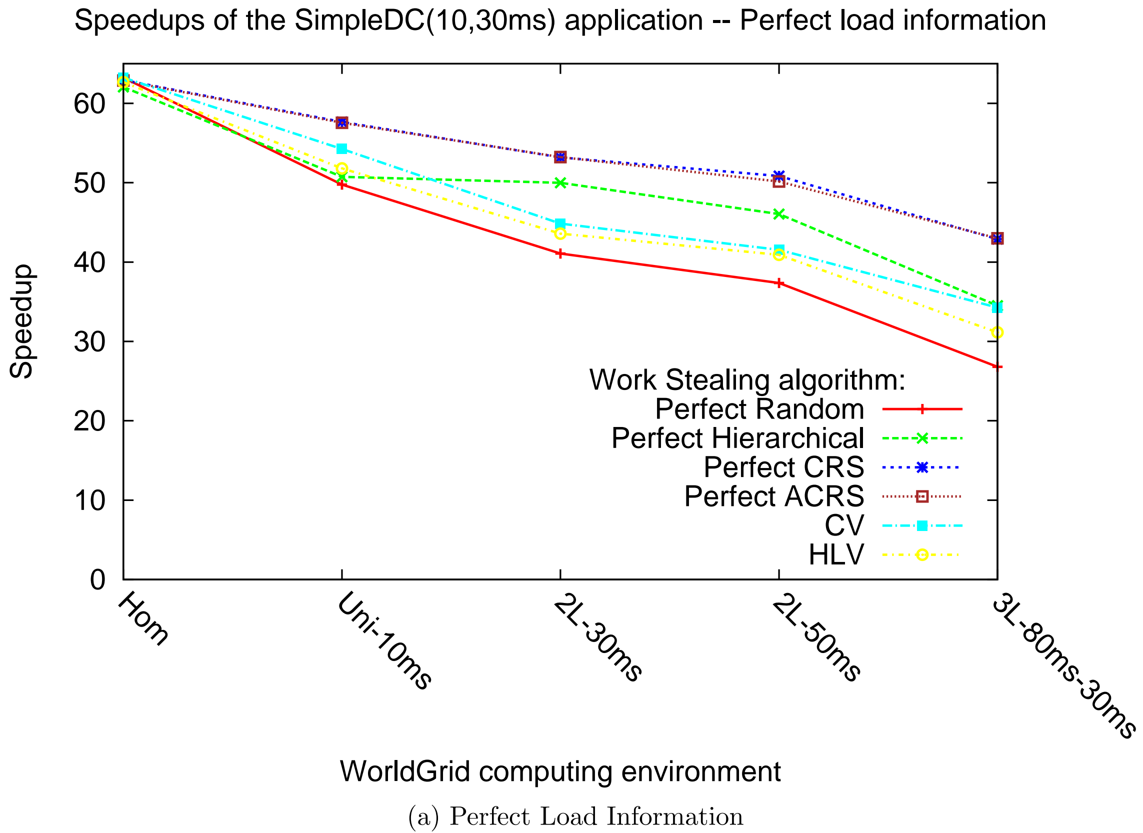
- As usual, the Perfect CRS algorithm gives the best speedups. For the applications with fine-grained tasks, Perfect CRS brings significant improvements over CRS. For applications with coarse-grained tasks, the improvements are minimal.

- Random Stealing can be significantly improved with the use of load information, both when sequential tasks are fine-grained and coarse-grained.

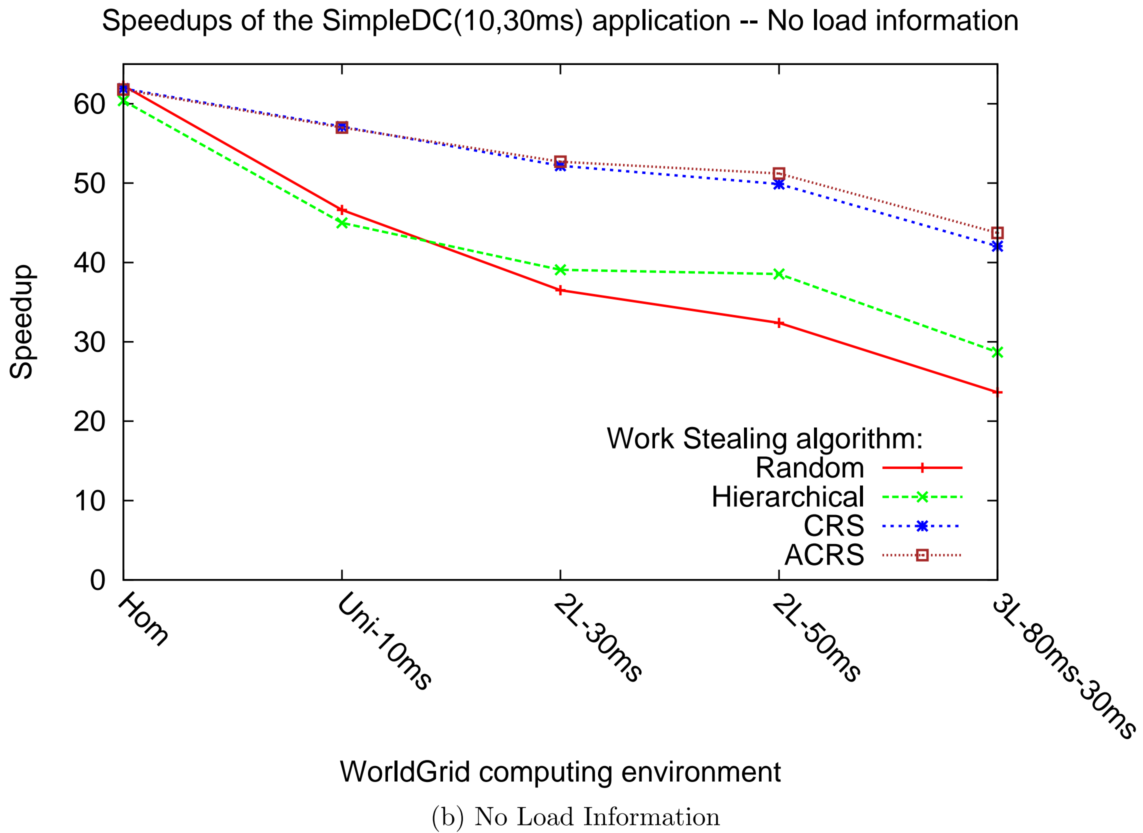
- The use of load information brings huge improvements (up to 60% in speedup) to Hierarchical Stealing when sequential tasks are fine-grained. When sequential tasks are coarse-grained, the improvements are smaller, but still significant (up to 30% in speedup).

Summary of the experiments with the SimpleDC applications

From all of the experiments with the SimpleDC applications, we can come to the following conclusions:



(a) Perfect Load Information



(b) No Load Information

Figure 5.11: Speedups of SimpleDC(10,30ms) application on the WorldGrid environments

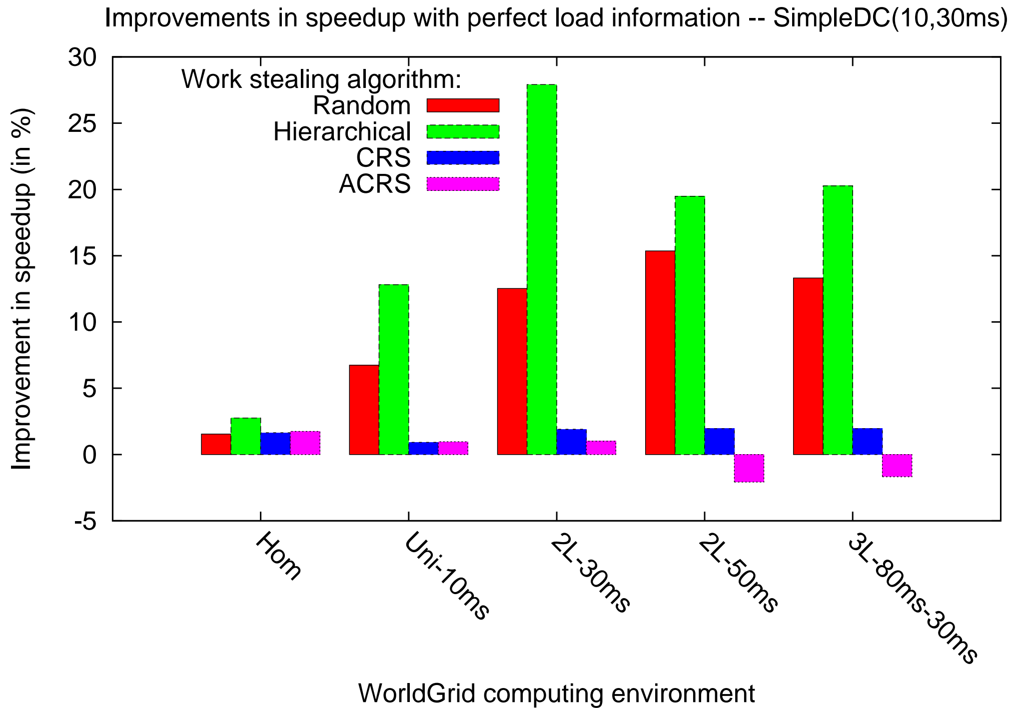


Figure 5.12: Improvements in speedups with perfect load information of the SimpleDC(10,30ms) application on the WorldGrid environments

1. The best speedups for applications of this type are obtained under the CRS algorithm, with the use of perfect load information.

2. If there is enough parallelism in application, or if sequential tasks are sufficiently coarse grained, there is not much benefit in the use of perfect load information in the CRS and ACRS algorithms. The basic CRS algorithm gives a very good speedups, and these speedups can only be marginally improved if the runtime system possesses perfect load information. In the case of applications with smaller amount of parallelism and with finer-grained sequential tasks, however, the use of perfect load information can increase the applications speedups up to 20% under the CRS algorithm on heterogeneous computing environments.

3. Random Stealing can be improved if the perfect load information is used for all applications on all heterogeneous computing environments. The improvements are more significant on large and highly heterogeneous computing environments, and for the applications that have smaller amount of parallelism.

4. Hierarchical Stealing can be significantly improved if load information is used,

as it enables thieves to avoid traversing large portions of PE-trees that do not have any work. Improvements that we can get are above 15% in almost all of the environments for all applications, and in some cases (applications with fewer number of parallel tasks, and where sequential tasks are fine grained) we observed the improvements of up to 55% in speedup.

5. For each considered algorithm, the improvements that perfect load information brings are better on larger environments, that consist of more clusters, or where clusters consist of more PEs. Also, the improvements are generally better on the environments with more heterogeneous communication latencies.

5.2.2 The DCFixedPar Applications

The previous section considered the applications that create a lot of parallelism (i.e. for which most of the tasks are nested-parallel). For these applications, we concluded that the most important thing for obtaining good speedups is to do local and remote stealing in parallel (the way it is done in the CRS algorithm). However, for the applications whose most of the tasks are sequential, this might not be the case. Due to a smaller number of nested-parallel tasks, work can be concentrated on fewer PEs during the execution of such applications. Furthermore, it is not the case any more that thieves will always steal large chunks of work. In fact, most of the times thieves are successful in locating work, they will manage to steal only sequential tasks, that may be fine grained. This means that, firstly, they will have to look for work more often and, secondly, they will not obtain enough work to also keep PEs near them busy (as was the case with the SimpleDC applications). Therefore, in order to obtain good speedups, it might be crucial to use load information to locate steal targets as quickly as possible, rather than looking for work blindly. We have seen an indication of this when we considered the SimpleDC(10,5ms) application.

In order to further investigate this, in this section we consider various DCFixedPar applications. We assume that the "divide" and "conquer" phases of all nested-parallel tasks are trivial and, furthermore, that all of the sequential tasks have the same size. We assume that these sequential tasks are fine-grained, so we will fix their size to $5ms$. Therefore, we will consider the DCFixedPar(n,k,5ms,0.1ms,0.1ms,l) applications, which we will denote just by DCFixedPar(n,k,l). The reasons for these simplifying assumptions are that we are interested in the amount of parallelism generated by different tasks, rather than the sizes of various sequential portions of these tasks.

If we fix the value of n, then for different values of k we get the applications with different number of tasks and different ratio between sequential and parallel tasks forked by each nested-parallel tasks. The larger k is, less nested-parallel tasks (and more sequential tasks) are forked by each nested-parallel task. Note that the larger the k is, less balanced the task-tree of an application is and, consequently, the higher the degree of irregularity an application is. Therefore, if the application X is DCFixedPar(n,k_1,L) and the application Y is DCFixedPar(n,k_2,L), "the application X is more irregular than the application Y" means the same as "the application X has a higher ratio of sequential to nested-parallel tasks", which in turns means the same as "k_1 is less than k_2".

DCFixedPar(40,k,4)

For our first set of experiments, we will consider DCFixedPar(40,k,4) applications, where k takes values from the set $\{3, 4, 5, 6, 7, 9, 11\}$[3]. The total number of tasks and the degree of irregularity of these applications is given in Figure 5.13.

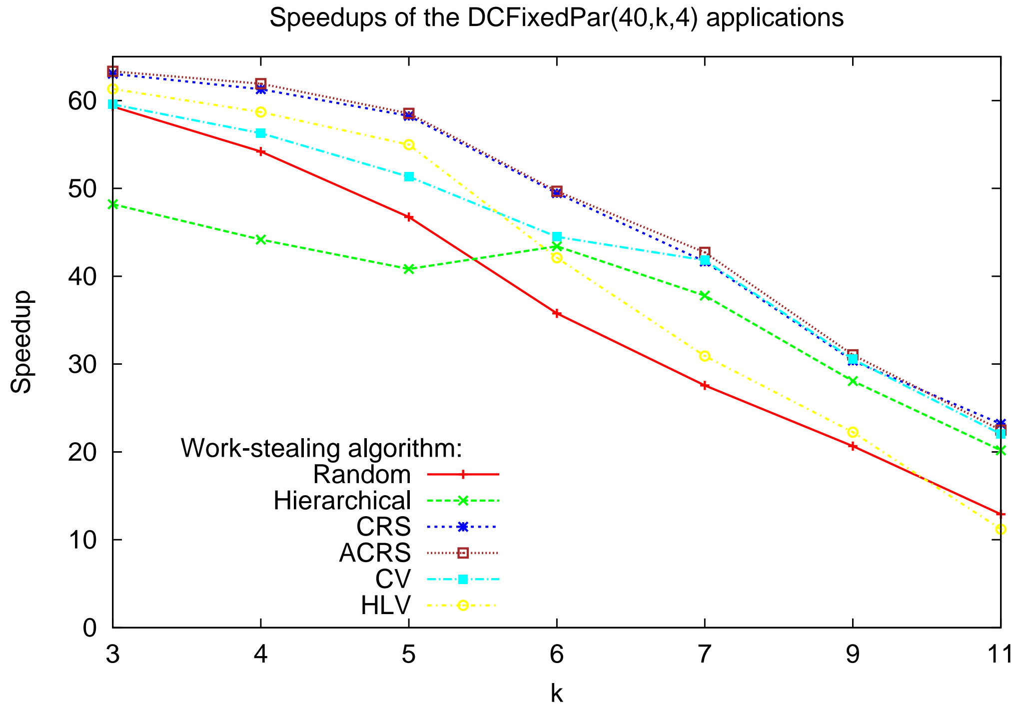
Figure 5.14 shows the speedups of the DCFixedPar(40,k,4) applications under considered algorithms on the WorldGrid-3L-80ms-50ms environment. Figure 5.14a shows the speedups when perfect load information is used, whereas Figure 5.14b shows the speedups without load information.

k	Number of tasks	The degree of irregularity
3	1237640	0.103284
4	444440	0.158401
5	187240	0.224318
6	62200	0.345560
7	31240	0.450674
9	13640	0.618618
11	4840	0.917055

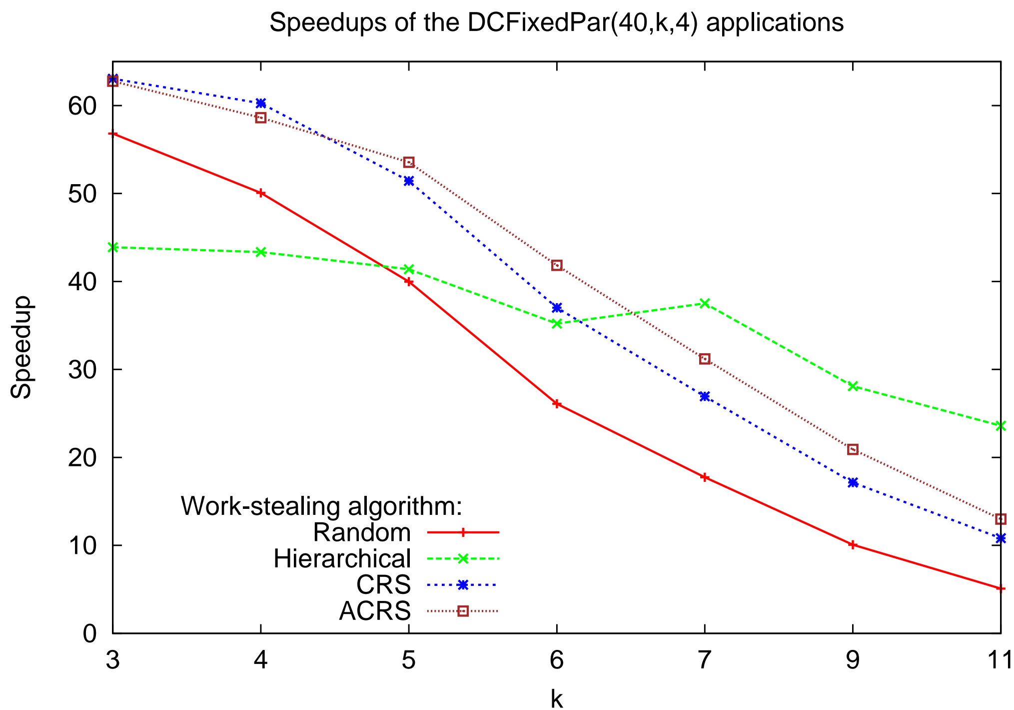
Figure 5.13: The number of parallel tasks and the degree of irregularity of the considered DCFixedPar(40,k,4) applications

From the Figure 5.14b, we can observe that, for more regular applications (k from 3 to 7), the ACRS algorithm performs the best when no load information is present, with CRS performing close to it. This is similar to what happens for the SimpleDC applications, as it shows the benefit of doing remote and local stealing in parallel.

[3]Note that DCFixedPar(40,7,4) and DCFixedPar(40,8,4) are the same applications, and the same holds for DCFixedPar(40,9,4) and DCFixedPar(40,10,4). This was the reason for omitting the cases where $k = 8$ and $k = 10$



(a) Perfect Load Information



(b) No Load Information

Figure 5.14: Speedups of the DCFixedPar(40,k,4) applications on the WorldGrid-3L-80ms-30ms Environment

However, for $k > 7$, we can see that Hierarchical Stealing starts to outperform both CRS and ACRS. This is due to the fact that, as k gets bigger, fewer nested-parallel tasks are there in the application and the parallel tasks are concentrated on fewer PEs. Therefore, we have the problem mentioned at the beginning of the section, where thieves are unable to obtain work locally most of the time, so they need to wait for the completion of remote steals. In most of the cases, CRS will behave similarly to Random Stealing[4]. Hierarchical Stealing systematically tries stealing from closer PEs to those that are farther and farther away from it and, in this situation, that leads to more efficient load distribution.

Figure 5.14b shows that, in the presence of perfect load information, the Perfect CRS and Perfect ACRS algorithms perform the best in all of the cases (even for larger k). CV also performs very good and close to the Perfect ACRS and Perfect CRS. In this case, there are no problems with idle PEs waiting longer than necessary to obtain work from remote PEs, as they know where to look for work. Additionally, we can observe a general trend of very similar performance of the CRS and ACRS algorithms, which indicates that preferring close targets for remote stealing does not bring much benefit if local and remote stealing are done in parallel. As opposed to this, if we compare the Random and CV algorithms, we can observe a notably better performance of CV. This shows that if random and local stealing are not done in parallel, then, as we can expect, there is notable benefit in choosing closer steal targets.

Figure 5.15 shows the improvement that the use of perfect load information brings to the speedups of considered applications on the WorldGrid-3L-80ms-30ms environment. In contrast to the SimpleDC applications, here we can observe very significant improvements for the Random, CRS and ACRS algorithms, especially for more irregular applications. We can see that, as the applications get more irregular, the more improvement we get with the use of load information, and for highly-irregular application where nested-parallelism is very sparse (e.g. for $k = 11$), we can observe excellent improvements in speedups of above 100%. For Hierarchical Stealing, we can observe very small and irregular improvements. This is due to a much better performance of Hierarchical Stealing that does not use load information for this kind of applications, especially for higher values of k. Where not much parallelism is available, Hierarchical Stealing is good at keeping most of the work locally (i.e. within the clusters nearby the main cluster), since the remote thieves are much slower in obtaining work from the main cluster. By the time the steal attempts from remote thieves reach the main

[4]Small difference is that in CRS when a target receives a steal from a remote thief and if it does not have any work to offload, it will forward the steal to some other target from its own cluster (as opposed to completely random target in Random stealing)

cluster, most of the (sparse) parallel work will already be distributed to the PEs from the main cluster, so remote thieves will probably obtain only smaller set of sequential tasks. In this case, it is a better solution than distributing tasks with nested parallelism to the remote thieves, since locality is lost in that way. Therefore, the fact that in Hierarchical Stealing a lot of steal attempts are fruitless actually helps to obtain a better application speedup.

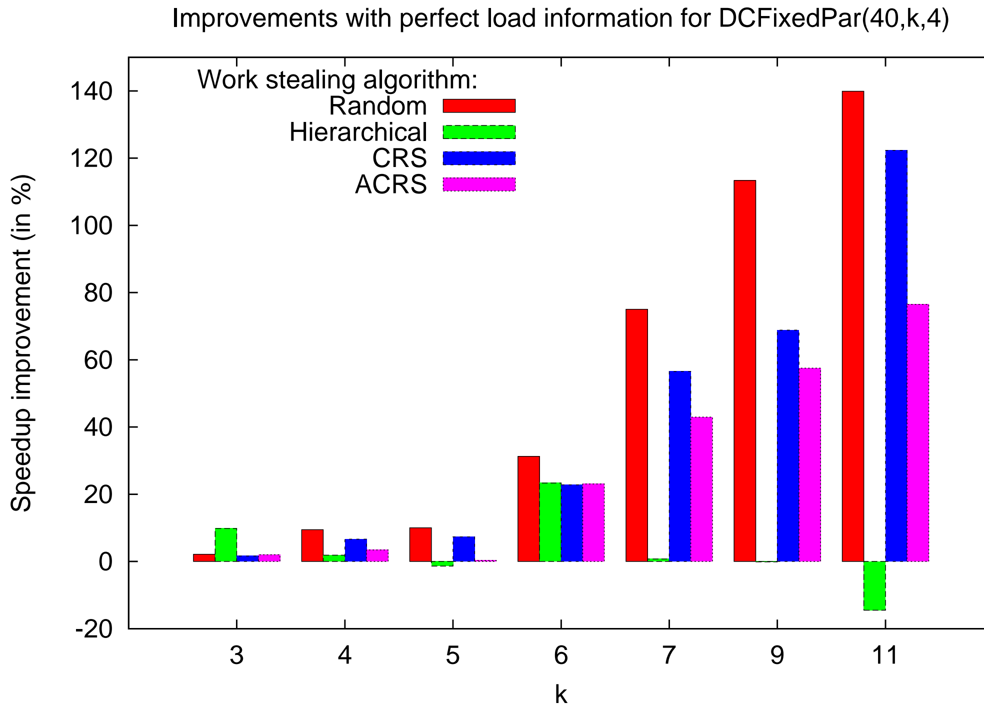


Figure 5.15: Improvements in speedups with perfect load information for the DC-FixedPar(40,k,4) application on the WorldGrid-2L-50ms environment

Overall, from Figures 5.14 and 5.15, we can see that on the highly heterogeneous computing environments, for the DCFixedPar(40,k,4) applications, the best of all considered algorithms are the Perfect CRS and Perfect ACRS algorithms. We can also see that, when there is enough parallelism in the application and where nested-parallel tasks are sparser, they give an excellent improvements in speedups (up to 120%), compared to the CRS and ACRS without load information. The Random Stealing and ACRS algorithms can also be very significantly improved with the use of load information, whereas the benefits for Hierarchical Stealing are negligible.

We have also observed that the Perfect CRS and Perfect ACRS algorithms give the best speedups on other heterogeneous computing environments for the DCFixedPar(40,k,4) applications. Therefore, the answer to the question of what algorithm performs the

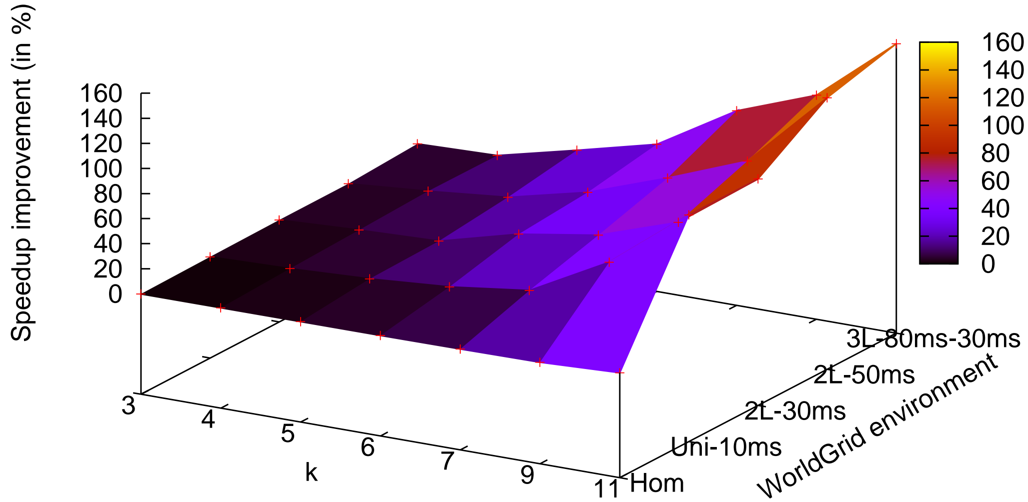
best for the DCFixedPar applications is the same as for the SimpleDC applications. Perfect CRS and Perfect ACRS algorithms are the ones to choose, and the difference between the two is minimal.

We will now focus on the improvements that perfect load information brings to the speedups of each algorithm individually for the DCFixedPar(40,k,4) applications on computing environments of increasing heterogeneity (the WorldGrid environments). Figure 5.16 shows the improvements for Random (Figure 5.16a) and Hierarchical (Figure 5.16b) algorithms on WorldGrid computing environments. For Random Stealing, we can clearly observe that the improvements are increasing as the degree of irregularity of application increases (i.e. as k increases), and as the heterogeneity of computing environment increases. The best improvements are obtained for $k = 11$ on the WorldGrid-3L-80ms-30ms environment. For the Hierarchical work stealing, however, we cannot observe a clear relationship between the amount of improvement obtained with perfect load information and the irregularity of the application/heterogeneity of the environment. In most of the combinations of application/computing environment, we can observe only very small improvements (and in some cases even a small decrease in speedup). However, for some applications with larger k on some environments, we can observe a very good improvements of up to 40% (for example, for the DCFixedPar(40,9,4) application on the WorldGrid-Uni-10ms environment).

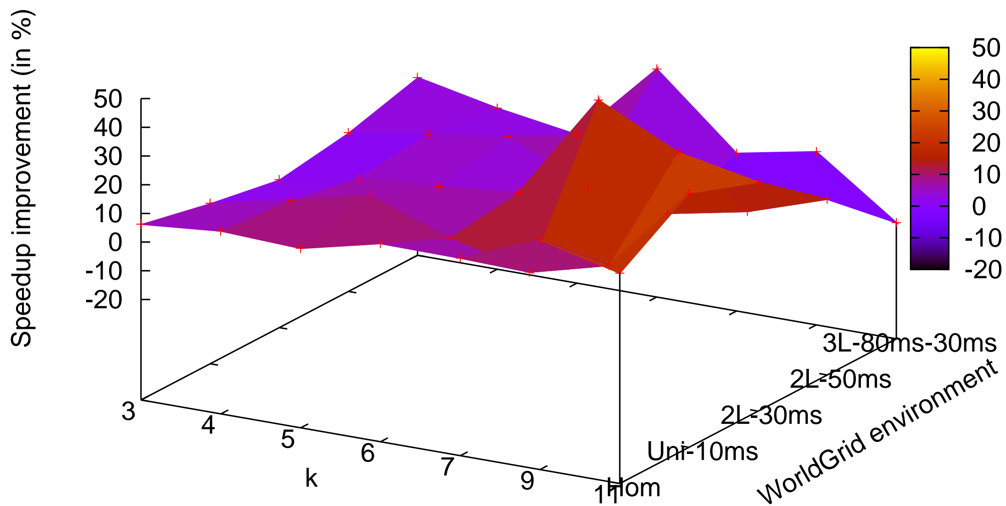
Figure 5.17 shows the improvements under CRS and ACRS algorithms. Here, we can observe the similar situation as for the Random stealing. The improvements are increasing as the degree of irregularity of an application increases and as the computing environments get more heterogeneous. Furthermore, we can observe that the improvements are most of the time in the range between 20-40% on more heterogeneous environments. However, on the highly heterogeneous environments for highly irregular applications (parameter k being greater than 7), we can observe the improvements in speedups in the range between 60 and 120%.

DCFixedPar(100,k,4)

As a final experiment in this section, we consider the DCFixedPar(100,k,4) applications, where k takes from the set $\{15, 20, 25\}$. Compared to the DCFixedPar(40,k,4) applications, these applications have bigger number of tasks, and the ratio between nested-parallel and sequential tasks is even smaller. So, a vast majority of tasks will be sequential, and only very small number will be nested-parallel, so we anticipate that the algorithms that do not use load information will run into big problems when trying to locate work. The number of tasks and the degree of irregularity of the considered

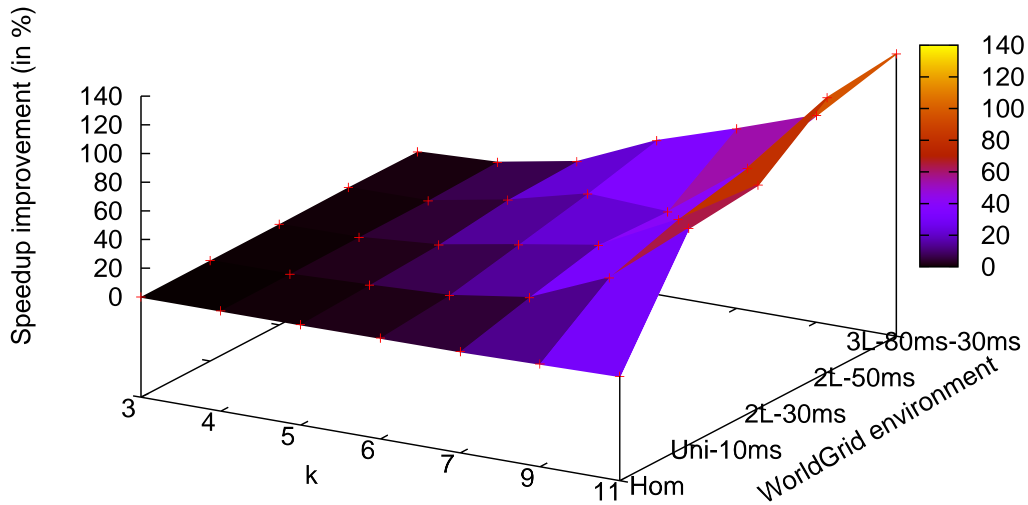


(a) Random Stealing

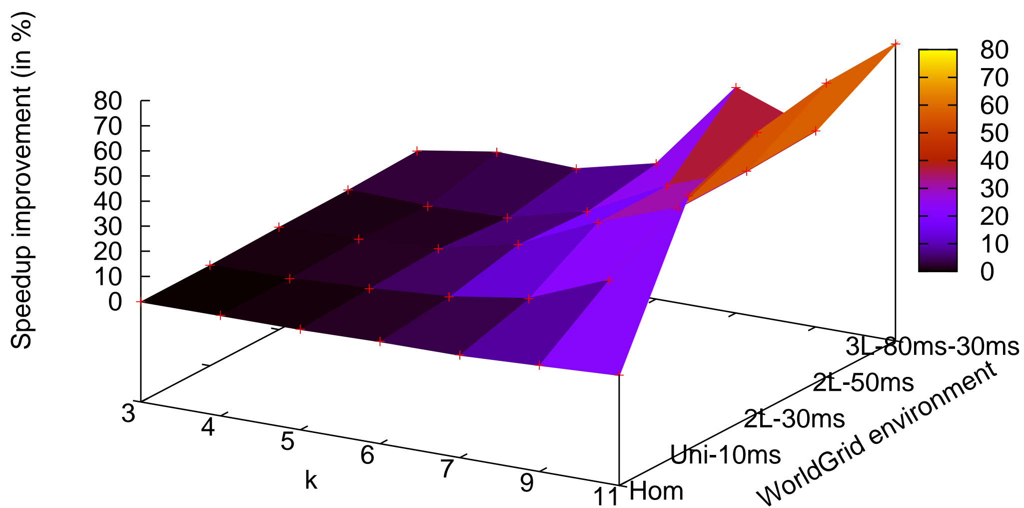


(b) Hierarchical Stealing

Figure 5.16: Improvements obtained with the use of perfect load information for the DCFixedPar(40,k,4) applications on the WorldGrid computing environments for Random and Hierarchical Stealing



(a) CRS



(b) ACRS

Figure 5.17: Improvements obtained with the use of perfect load information for the DCFixedPar(40,k,4) applications on the WorldGrid computing environments for CRS and ACRS

applications is showed in Figure 5.18.

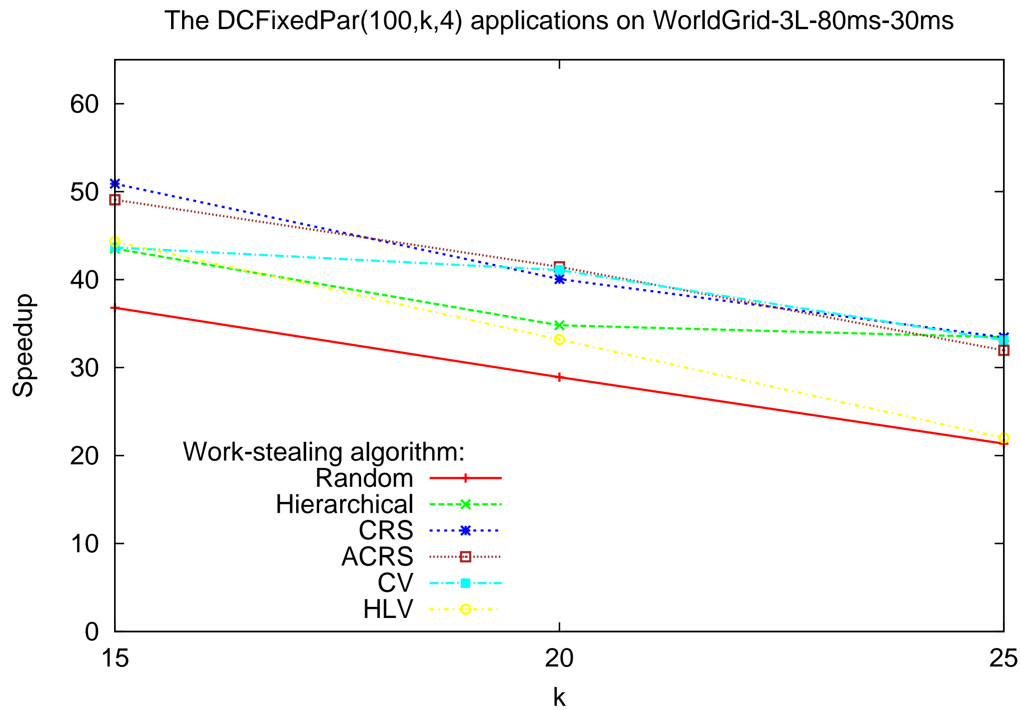
k	Number of tasks	The degree of irregularity
15	155500	0.575651
20	78100	0.744058
25	34100	1.012648

Figure 5.18: The number of parallel tasks and the degree of irregularity of the considered DCFixedPar(100,k,4) applications

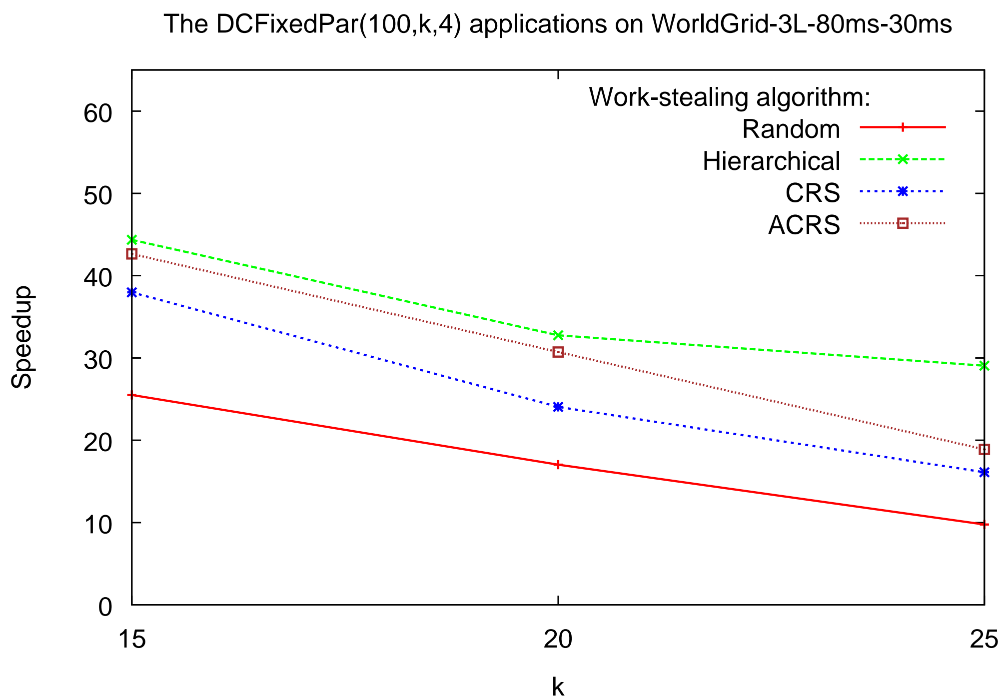
Figure 5.19 shows the speedups of the DCFixedPar(100,k,4) applications on the WorldGrid-3L-80ms-30ms computing environment. We can observe similar results as for DCFixedPar(40,k,4) applications. Again, Perfect CRS and Perfect ACRS give the best speedups of all algorithms that use perfect load information, with CV being very close to them. Looking at the algorithms that do not use load information (Figure 5.19b), we can observe that Hierarchical Stealing gives the best speedup for all applications. The reasons for this are similar as for the more irregular DCFixedPar(40,k,4) applications.

Figures 5.20 and 5.21 show the improvements that the use of perfect load information brings to the speedups of the considered DCFixedPar(100,k,4) applications on various WorldGrid computing environments for all considered algorithms. Similarly to the DCFixedPar(40,k,4) applications, we can observe that for the Random, CRS and ACRS algorithms the amount of improvement increases as the k increases, and as the heterogeneity of computing environment increases. We can also observe similar improvements for the DCFixedPar(100,k,4) and DCFixedPar(40,k,4) applications with the similar degree of irregularity. For example, DCFixedPar(100,15,4) and DCFixedPar(40,7,4) have similar degrees of irregularity, and the improvements under Perfect CRS for both applications increase from 0 to about 40%, as the heterogeneity of the environment increases.
For Hierarchical Stealing, on the other side, we can again observe more or less random improvements, although improvements are generally better for larger k and more heterogeneous environments.

From the experiments with the DCFixedPar(100,k,4) applications, we can get more or less the same conclusions as for the DCFixedpar(40,k,4) ones. Perfect CRS and Perfect ACRS give the best speedups (with CV being close to them for more irregular applications), speedups for almost all algorithms can be increased when perfect load information is used, and the improvements are better for more irregular applications executed on more heterogeneous computing environments.

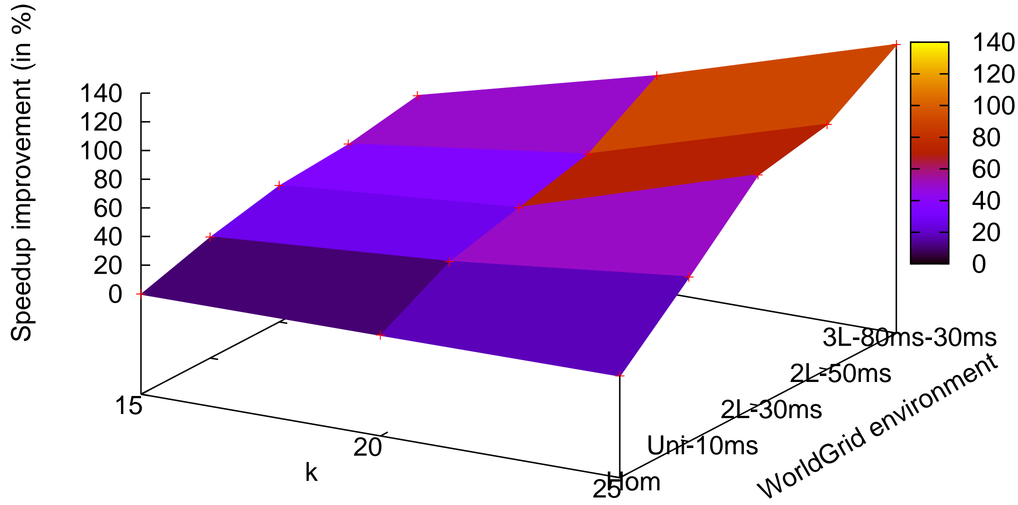


(a) Perfect Load Information

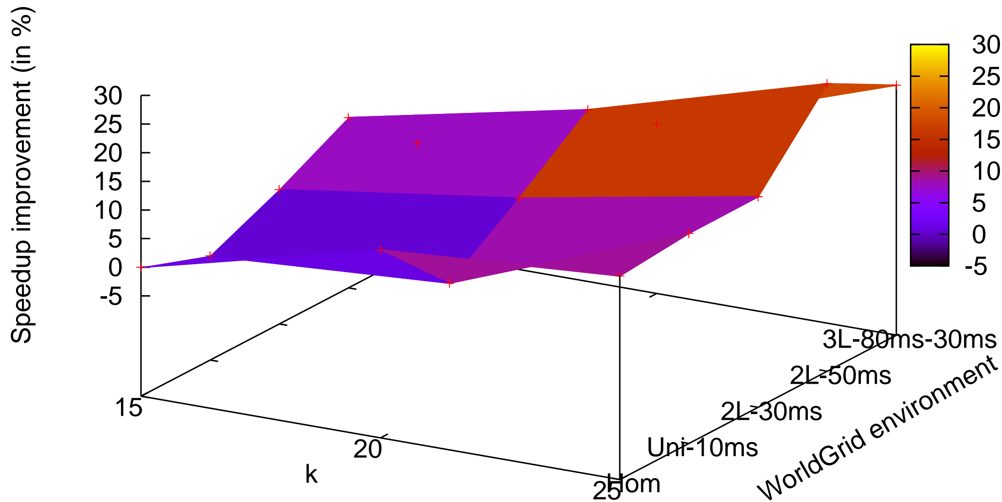


(b) No Load Information

Figure 5.19: Speedups of the DCFixedPar(100,k,4) applications on the WorldGrid-3L-80ms-30ms Grid Environment

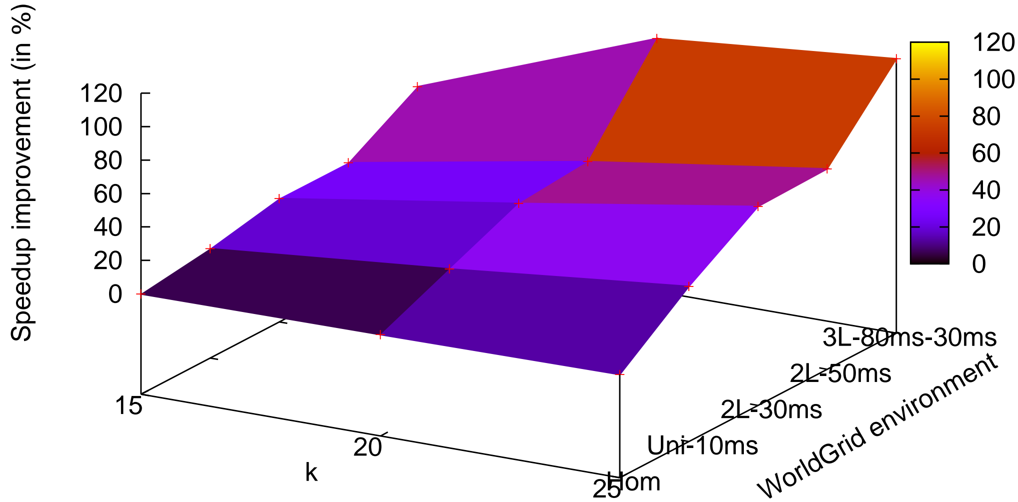


(a) Random Stealing

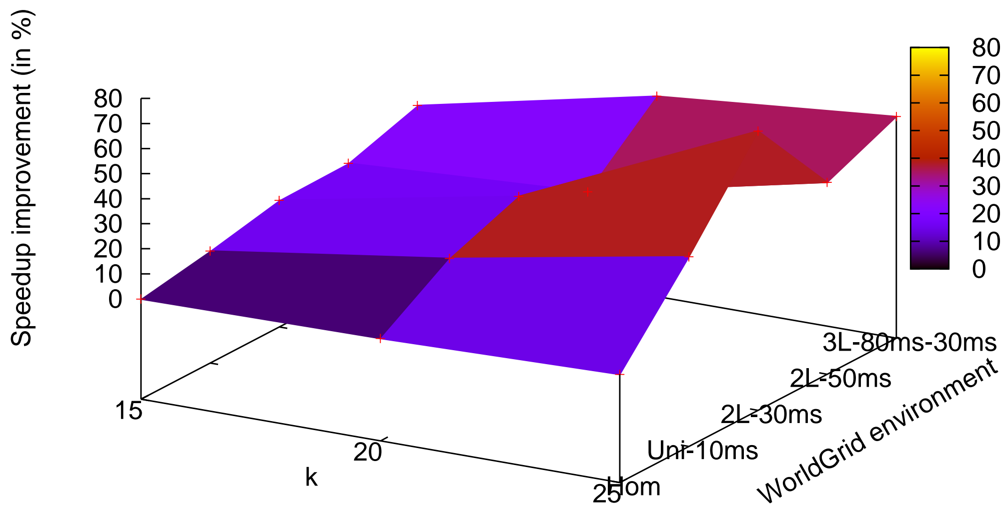


(b) Hierarchical Stealing

Figure 5.20: Improvements obtained with the use of perfect load information for the DCFixedPar(100,k,4) applications on the WorldGrid computing environments for Random and Hierarchical Stealing



(a) CRS



(b) ACRS

Figure 5.21: Improvements obtained with the use of perfect load information for the DCFixedPar(100,k,4) applications on the WorldGrid computing environments for CRS and ACRS

Summary of the experiments with DCFixedPar applications

From the experiments with the DCFixedPar applications, we can make the following conclusions about the irregular applications in which most of the tasks are sequential, and which have smaller number of tasks that create additional parallelism:

- When load information is used, the Perfect CRS algorithm gives the best speedups on all computing environments we considered.

- When no load information is used, CRS and ACRS give the best speedups for more regular applications, whereas Hierarchical Stealing is the best for highly irregular applications, as it manages to keep most of the work local to the main cluster.

- Speedups under the Random, CRS and ACRS algorithms can be significantly improved with the use of load information.The improvements for these algorithms are better the more irregular the application is, and the more heterogeneous underlying computing environment is. For highly irregular applications on highly heterogeneous computing environments, we have observed the improvements of up to 160%, 140% and 80% for Random, CRS and ACRS algorithms, respectively.

- For Hierarchical Stealing, we did not manage to establish the relationship between the amount of improvement that the use of load information brings, and the irregularity of an application/heterogeneity of an environment. However, in most of the cases, we can observe at least some improvements, but also some cases exist where the use of load information actually decreases the performance of Hierarchical Stealing.

5.2.3 Summary of Experiments

The experiments in Sections 5.2.1 and 5.2.2 evaluated what is the best way to use load information in work-stealing, for what applications on what computing environments are the benefits of its use the biggest, and how much the individual state-of-the-art work-stealing algorithms can benefit from this information.

We focused on the SimpleDC and the DCFixedPar applications (described in Section 4.2), as our main goal was to investigate the use of load information for applications that have variable amount of parallelism and variable ratio between sequential and nested-parallel tasks. the SimpleDC applications are the examples of very regular

applications that generate a lot of parallelism, and where most of the tasks are nested-parallel (and which, therefore, they generate further parallelism). The DCFixedPar applications, on the other hand, can be highly irregular and, although they generate a lot of parallelism, vast majority of their tasks can be sequential. Therefore, the number of potential steal victims in the DCFixedPar applications at each point of its execution is typically much smaller than for the SimpleDC applications and, therefore, algorithms that do not use load information might struggle in locating them.

We have considered the execution of such applications on various computing environments, from fully homogeneous (WorldGrid-Hom) to very heterogeneous (WorldGrid-3L-80ms-30ms) environments.

Experiments in this section verified the hypothesis that the use of load information can significantly improve the speedups of irregular applications, that have much more sequential then nested-parallel tasks and where parallel work is concentrated on fewer PEs.

Specifically, we made the following conclusions:

1. Whereas in the case of algorithms that do not use load information, there is no clear 'winner' for all considered applications (i.e. no algorithm gives the best speedup for all applications on all computing environments), when perfect load information is present, Perfect CRS and Perfect ACRS clearly outperform all other algorithms for all of the combinations of applications and computing environments. We can, therefore, conclude that the CRS or ACRS algorithms should be used for work-stealing of the applications with tree-like structure of tasks, provided that we can find a way for runtime system to have a good approximation of the PE load. We have also observed that the performance of CRS and ACRS algorithms is very similar, so exactly which one of them to use is not very important. Since the implementation of CRS is simpler, and it does not rely on the monitoring of the performance of network links, we can recommend using CRS.

2. The use of perfect load information brings the most benefits to the execution of parallel applications which generate significantly larger number of sequential than nested-parallel tasks. If most of the application's tasks have nested parallelism (for example, in typical divide-and-conquer applications), then the amount of parallelism created is large enough to keep all of the PEs busy, and the work can be efficiently distributed with work-stealing methods that do not use any load information.

3. For the applications that generate significantly larger number of sequential tasks than nested-parallel tasks, the use of load information brings the most significant improvements under all algorithms on the highly heterogeneous computing environments. The only exception is Hierarchical work-stealing, where there is no clear correlation between the amount of improvement and the heterogeneity of the computing environment.

   For the applications which most of the tasks are nested-parallel (the SimpleDC applications), significant improvements can only be observed on large computing environments (i.e. those comprising a large number of clusters, or where each cluster comprise a large number of PEs). The exception is, again, Hierarchical Stealing, where significant improvements for this kind of applications can be observed on all computing environments.

5.3 Feudal Stealing

Our experiments in the previous section indicated that the use of load information can significantly improve the applications' speedups under work-stealing algorithms. The improvements were especially significant for more irregular parallel applications on heterogeneous computing environments. We have also seen that the CRS algorithm with perfect load information delivers the best speedups out of all considered algorithms for all applications that we tested, on all computing environments. Of course, in realistic runtime systems, the PEs will not have perfect load information. Instead of that, each PE has the perfect information only about its own load. The PEs, therefore, need to exchange load information in some way, so that each of the PEs has at least a good approximation of the load of other PEs.

In this section, we describe a novel Feudal Stealing algorithm, which addresses the problem of PEs obtaining an accurate load information. This algorithm uses the same principle for choosing steal targets as the CRS algorithm, namely, it does local and remote stealing in parallel. Feudal Stealing extends the CRS algorithm with a combination of *locally-centralised* and *remotely-distributed* mechanisms for exchanging information about the PE loads. In our case, locally-centralised mechanism means that the load information is centralised within each cluster. That is, in each cluster there is a *head PE* that holds the information about the load of all other PEs from that cluster. Each PE periodically reports its load to its head PE (like in feudalism, hence the name of the algorithm). Remotely-distributed mechanism means that head PEs exchange their approximations of the cluster loads in a peer-to-peer way, so there

is not a *global head* PE. The way in which load information is distributed between head PEs is very similar to the Grid-GUM mechanism (see Section 2.4.3).

As described above, in Feudal Stealing, in each cluster we nominate one PE to be that cluster's head PE. The responsibility of the cluster head PE is to route all the steal attempts that come inside its cluster or that go outside of it. Therefore, each steal attempt whose target is outside of the thief's cluster needs to be sent via the head PE of the thief's cluster. Similarly, each steal attempt sent outside of the target's cluster needs to pass through the target's cluster head PE. In order to be able to route steal attempts appropriately, head nodes hold information about the loads of all PEs within their clusters, as well as the approximation of the loads of all remote clusters.

Within each cluster, each PE periodically sends its load information to its head PE. This information is sent whenever the load of the PE changes (i.e. whenever the length of its task pool changes). Alternatively, to prevent the case where the head PE is flooded with the messages notifying it of the changes in PE loads (for applications with fine-grained tasks), we introduce the minimal amount of time between the two successive load information messages sent from one PE [5]. In this way, the head PE will have more-or-less accurate information about the load of individual PEs from its cluster.

Assuming that all cluster head PEs hold the approximation of the load of all other clusters, work-stealing proceeds in the following way. As in the CRS algorithm, each PE has the `remoteStealing` flag (initially set to `false`), which denotes whether the PE is currently doing remote stealing or not. Whenever a PE becomes idle, it starts looking for work locally (within the same cluster) and remotely (if `remoteStealing` flag is not set) in parallel. When a PE starts looking for work remotely, it sets its `remoteStealing` flag to `true`.

Local stealing is done in the same way as in the CRS algorithm, namely, randomly without using the head PE's load information. Remote stealing proceeds in the following way. Assume that thief t_0 from cluster C_0 is looking for work remotely. t_0 first sends the remote-steal message to the head PE h_0 of cluster C_0. h_0 then checks its its approximation of the load of other clusters, and based on it chooses the head PE h_1 of some remote cluster C_1 and forwards the remote-steal message to it (see Figure 5.22). Once h_1 receives this message, it checks the load of PEs from its own cluster (cluster C_1). Depending on this load, two situations can happen:

- PE v_1 with non-zero load exists in cluster C_1. The remote-steal message is

[5] This is, of course, only relevant to the implementation of the algorithm in the real runtime systems, as in simulations we ignore the overheads in processing messages, and, therefore, overloading of head node with messages cannot occur.

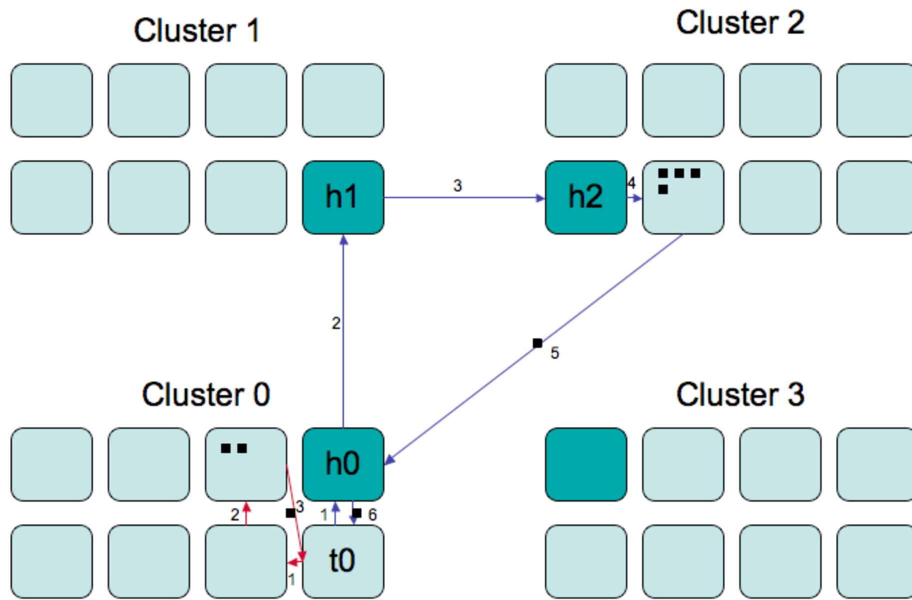


Figure 5.22: Sequences of local and remote steal messages exchanged in Feudal Stealing. Blue arrows denote remote-steal messages, whereas the red ones denote local-steals. Black squares denote tasks that PEs hold, and the arrow with black square denotes the steal message that is carrying a task.

then forwarded to v_1, which becomes a victim. v_1 responds to the remote-steal message by sending a task to h_0, which then forwards it to the thief t_0. The task is sent via h_0 (and not directly to t_0) for the reasons that will become clear later.

- All PEs from cluster C_1 have zero load. In this case h_1 checks its approximation of the loads of other clusters, and chooses the head PE h_2 of some cluster C_2. It then forwards the remote-steal message to h_2.

The stealing proceeds in this way until either a PE with non-zero load is found, or some predefined number of clusters is visited. In the latter case, the remote-steal message is returned back to thief t_0 (via head PE h_0), which then sets its `remoteStealing` flag to `false`.

The way in which a head PE selects the remote cluster to which to send the remote-steal message depends on its approximation of the load of other clusters and the load of a cluster where the remote-steal message was initiated from. Assume that the load of the cluster where remote-steal was initiated from is l. The three cases can happen:

1. At least one cluster with load higher than l exist. Assume that $\{C_1, C_2, \ldots, C_n\}$ is the set of all clusters whose load is higher than l. Then one of these clusters

is chosen randomly, and the probability that cluster C_i is chosen is proportional to its load.

2. No cluster with load higher than l exists, but one or more clusters with non-zero load exist. Assume $\{D_1, D_2, \ldots, D_n\}$ is the set of all clusters with non-zero load. Then one of the clusters from this set is chosen randomly, where, again, the probability that cluster D_i is chosen is proportional to its load.

3. All clusters have zero load. In this case, choose a random cluster.

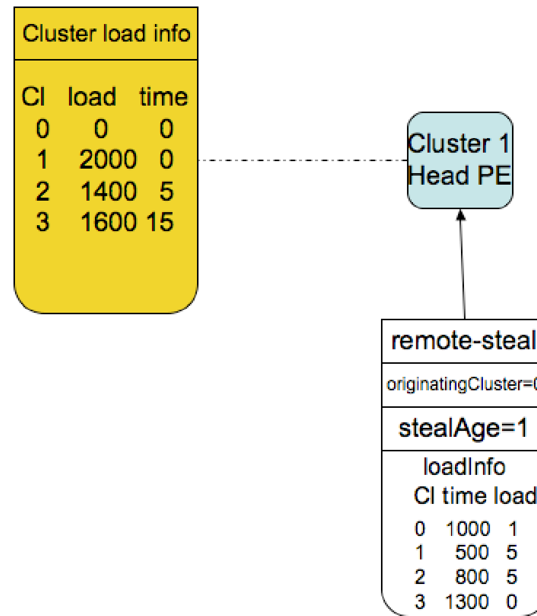
In the first two cases, the probability that a cluster is selected is proportional to its load. For example, if the head PE of the cluster has information that the load of cluster C_1 is 20, and the load of cluster C_2 is 100, then the probability that cluster C_1 is chosen is 20/120, and that cluster C_2 is chosen is 100/120.

The head PEs update their load information in a similar way it is done in the Grid-GUM algorithm. Every remote-steal message contains, for each cluster in the environment, the approximation of its load, together with the timestamp when this approximation was obtained. Each head PE that receives such a message compares the timestamps from the message with the timestamps of its own approximations of loads of other clusters. If, for some cluster, the approximation in the message is more recent, the head PE updates its own approximation (together with its timestamp) for this cluster. Conversely, if, for some cluster, the approximation that PE has is more recent that the one from the message, then the approximation from the message (together with its timestamp) is updated. This is illustrated on Figure 5.23. Initially, when remote-steal message is initiated from the thief, timestamps for the approximation of each cluster are set to 0.

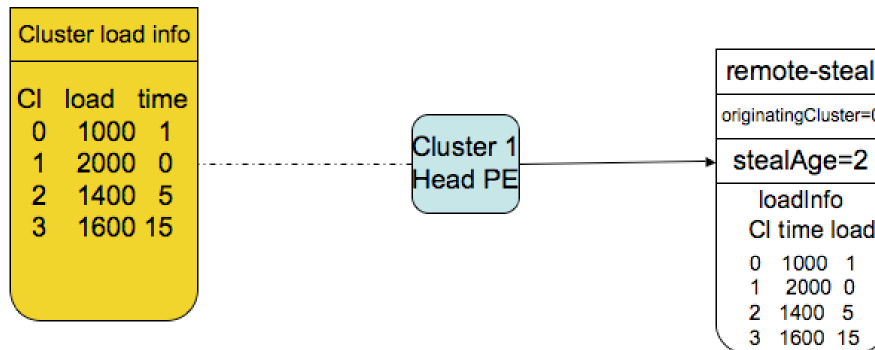
The mechanism of attaching the load approximation to the remote-steal messages enables head PEs to have more accurate information about loads of other clusters. Also, this is the reason why a victim does not send the task directly to the thief, but rather to the head PE of its cluster. Sending the task to the head PE enables it to update its load approximation from the message. The overhead in sending the task via the head PE is just in sending one additional message within a cluster. Since we assume that the communication latency with clusters is low, this overhead is not high.

Comparison Between Feudal and Grid-GUM Work-stealing

Since Feudal work-stealing bears many similarities with the Grid-GUM algorithm, it is natural to compare these two algorithms. We can note several advantages of Feudal algorithm:



(a) On receiving the remote-steal



(b) After processing the remote-steal

Figure 5.23: Updating the approximation of load info in the remote-steal message, and in cluster head PE

- *Smaller steal messages* – In Grid-GUM, *each* steal message contains the load of *every* PE in the environment. In Feudal Stealing, load information is attached only to the messages related to the remote stealing. Furthermore, only the information about loads of the entire clusters (and not of the individual PEs from them) is attached to the message. This has an important advantage in that the steal messages are generally smaller under Feudal Stealing, especially on large-scale computing environments that comprise of a large number of PEs. In these environments, the steal messages that Grid-GUM sends may be prohibitively large, since they contain the information about the load of every PE in the environment. This might be an issue when such messages are sent over network links with limited bandwidths.

- *Less sensitive to the small changes in loads of individual PEs* – In Feudal Stealing, the head PEs only hold the information about the loads of the entire clusters (as opposed to the Grid-GUM mechanism, where all PEs holding the information about the load all other PEs). In other words, the decisions about from where to steal in Feudal Stealing are based on the approximation of the load of entire clusters, and, by sending the remote-steal message to the cluster's head PE, stealing is attempted from the cluster *as a whole* (rather than from the individual PEs from it). This makes the stealing less sensitive to the changes in load of individual PEs.

  For example, if the load of the PE p (from the cluster P) drops to zero in Grid-GUM, many thieves might not be informed of this immediately and, therefore, they might unsuccessfully attempt stealing from p. In Feudal Stealing, on the other hand, there may still be other PEs from P with non-zero load. Therefore, if a thief tries to steal from the cluster P (because it assumes that it has non-zero load), the problem only appears if the drop in load of p made the load of the whole cluster to be zero.

- *PEs cannot get isolated* – In Feudal Stealing, each PE at regular intervals reports its load to the head PE of its cluster. This eliminates the possibility of a PE becoming isolated if it does not communicate with the rest of the environment. Consequently, there is less chance for the PEs to have seriously outdated information about the load of some PE.

- *More accurate load information* – In Feudal Stealing, only head PEs hold the load information. This means that less PEs need to exchange load information and, therefore, higher is the chance that these PEs will have an accurate load

information. In addition, since each remote stealing message needs to pass via at least two head PEs, head PEs will very frequently receive these messages, and will have their load information updated very frequently.

Of course, nothing comes for free. The most important drawback of Feudal Stealing, compared to Grid-GUM, is its semi-centralised nature, which makes it less fault-tolerant. Additionally, more communication is involved in Feudal Stealing, which makes it more prone to central PEs being overloaded with communication. This is the price that needs to be paid in order to have more accurate load information.

5.3.1 The Feudal Stealing Algorithm

Algorithms 15 - 20 give the pseudocodes for the most important parts of the Feudal Stealing algorithm. Algorithm 15 shows the pseudocode for the `peIdle` function, which is executed whenever a PE becomes idle. Idle PE sends a local-steal message with `age` parameter set to `0` to some local PE (chosen via a call to `choosePE()` function) (lines 17–23). If the idle PE does not have any outstanding remote-steal messages, it also sends a remote-steal message to its cluster head PE (which will route it appropriately) (lines 1–16). Of course, if the idle PE itself is the head PE, it immediately sends the message to a head PE of some remote cluster, and attaches to it its approximations of the loads of all clusters and the timestamps when these were obtained (lines 9–13).

In the discussion below, we will refer to the PE that created the remote-steal message as the *thief that initiated the remote-steal message*

Algorithms 16 – 18 show the pseudocodes for the functions executed when a PE receives the steal message. Note that there are two kinds of steal messages – one that contains a task as a part of it (equivalent to the `SCHEDULE` message in Grid-GUM) and the one that does not. The steal message that does not contain a task (i.e. whose `task` attribute is `NULL`) is the one looking for work. Steal message with the task attached to it is the one that has found work, and needs to be returned to the thief who initiated it.

Algorithm 16 shows what happens when a PE p receives a local-steal message. The function is split into the three cases:

1. The local-steal message was initiated by p (lines 1–4). This means either that the message has found some work (in which case, the work is added to the p's task pool), or that no work has been found after visiting `MAX_LOCAL_AGE` PEs. In either case, the message does not get forwarded any further.

2. The local-steal message was not initiated by p, and p has some tasks in its task pool (lines 5–8). In that case, p chooses one task from its task pool, attaches it to the local-steal message and forwards this message to the thief that initiated it

3. The local-steal message was not initiated by p, and p does not have any tasks in its task pool (lines 10–15). p then first increases the age of the steal message, and then checks whether the age exceeds the maximal allowed age for local-steal messages (`MAX_LOCAL_AGE`). If it does, the message is returned to the thief that initiated it. Otherwise, the message is forwarded to some random PE from the same cluster.

Note that in the algorithm, we have assumed that a single global parameter

Algorithm 15 Feudal Stealing – peIdle(PE pe) function

```
1: if (!remoteStealing) then
2:    remoteSteal = new (StealMessage)
3:    remoteSteal.type = REMOTE_STEAL
4:    remoteSteal.originatingCluster = pe's cluster
5:    remoteSteal.originatingPE = pe
6:    remoteSteal.age = 0
7:    remoeSteal.task = NULL
8:    if (pe is a head PE of its cluster) then
9:       nextTarget = choosePE (REMOTE)
10:      for all clusters do
11:         remoteSteal.loadInfo[cluster].time = pe.loadInfo[cluster].time
12:         remoteSteal.loadInfo[cluster].load = pe.loadInfo[cluster].load
13:      end for
14:   else
15:      nextTarget = cluster's head PE
16:   end if
17:   send remoteSteal to nextTarget
18:   remoteStealing = true
19: end if
20: localSteal = new (StealMessage)
21: localSteal.type = LOCAL_STEAL
22: localSteal.age = 0
23: localSteal.originatingPE = pe
24: localSteal.task = NULL
25: localTarget = choosePE (LOCAL)
26: send localSteal to localTarget
```

MAX_LOCAL_AGE exists for all clusters, which denotes the maximal amount of hops the the steal message is allowed to travel before being returned to the thief that initiated it. The purpose of this parameter is to prevent overloading the environment with steal messages in the situation where no extra work exist on any PE. In the implementations of Feudal Stealing, MAX_LOCAL_AGE can, rather than a constant, be a function of cluster size, so that more hops are allowed to be made for larger clusters.

Algorithm 16 Feudal Work Stealing – processLocalStealMessage(PE p, StealMessage message) function for head and non-head nodes

```
 1: if message.originatingPE == p then
 2:    if message.task != NULL then
 3:       add message.task to p's task pool
 4:    end if
 5: else if p has tasks in its task pool then
 6:    taskToSend = choose a task from p's task pool
 7:    message.task = taskToSend
 8:    send message to message.originatingPE
 9: else
10:    message.age++
11:    if message.AGE < MAX_LOCAL_AGE then
12:       nextTarget = choosePE (LOCAL)
13:    else
14:       nextTarget = message.originatingPE
15:    end if
16: end if
```

The way in which a remote-steal message is processed depends on whether the PE who receives it is a head PE of a cluster or not. Algorithm 17 shows what happens when the non-head PE p receives a remote-steal message. A non-head PE can receive remote-steal message only from the head PE of its cluster. We can see that the code is similar to the case where local-steal message is received. Again, there are three cases:

1. The remote-steal message was initiated by p (lines 1–5). This case is the same as when local-steal message is received, except that also `remoteStealing` flag is set to `FALSE`.

2. The remote-steal message was not initiated by p, and p has some tasks in its task pool (lines 6–9). Again, this case is almost the same as for local-steal message, except that the message is forwarded to the head PE of the thief initiated it, rather than directly to the thief.

3. The remote-steal was not initiated by p, and p does not have any task in its task

pool (lines 10–11). In this case, the message is forwarded to the head PE of p's cluster. We can see that here, as opposed to the case where local-steal message is received, the `age` parameter of the message is not increased. Whereas for local-steal messages, the `age` parameter denotes the number of PEs visited by that message, for remote-steal it denotes the number of *clusters* visited. Therefore, only when remote-steal message is forwarded from one cluster to the other is the `age` attribute increased.

One can ask how can the case 3 even happen, if the remote-steal message was forwarded to p from its head PE. Head PE that receives the remote-steal message is supposed to forward it to some PE from its cluster only in the case that this PE has some work to send. In the case 3, however, the remote-steal message was forwarded to the PE that does not have any work. This can happen if the head PE has inaccurate information about the load of the PE from its cluster, so it chooses p under the wrong assumption that it has work to offload.

Algorithm 17 Feudal Work Stealing – processRemoteStealMessage(PE p,StealMessage message) function for non-head nodes

```
1: if message.originatingPE == p then
2:    remoteStealing=false
3:    if message.task != NULL then
4:       add message.task to the p's task pool
5:    end if
6: else if p has tasks in its task pool then
7:    taskToSend = choose a task from the p's task pool
8:    message.task = taskToSend
9:    send message to the head PE of p's cluster
10: else
11:    send message to the head PE of p's cluster
12: end if
```

The case when a PE that received the remote-steal message is a head PE of a cluster is more complicated. This is due to the fact that all remote-steal messages pass through the cluster head PEs (in order for the remote-steal messages and head PE's approximation of load information to be updated), even if the head PEs are not directly involved in steal operation. Therefore, as opposed to the non-head PEs, the head PEs need also to act as routers both for messages that have some work attached to them, and also for the messages that are still looking for work. Pseudocode for this case is given in Algorithm 18, where the head PE p receives the remote-steal message. Firstly, the approximation of the load information from the message and

the approximation that p owns are synchronised. This is done in `updateLoadInfo()` function. After that, five cases are separated:

1. The remote-steal message was initiated by p (lines 1–6). This case is the same as for the non-head PEs.

2. The remote-steal message was not initiated by p, but the message age is above the allowed limit (lines 7–8). This means that the message is on its way to the thief who initiated it. Since as soon as the message age limit gets reached, the message is forwarded to the head PE of the thief's cluster (see case 5), it must be that p is from the same cluster as the thief. Therefore, p just needs to forward the remote-steal message to the thief who initiated it.

3. The remote-steal message was not initiated by p, the message did not reach its age limit, and the message contains a task (lines 9–14). This is the case where the message is on its way to the thief who initiated it. This also means that p is the head PE of the thief's cluster, so p just forwards the message to the thief.

4. The remote-steal message was not initiated by p, the message did not reach its age limit, it does not contain a task and p has some tasks in its task pool (lines 15–18). In this case, one of the tasks is selected and attached to the message, and the message is sent to the head PE of the thief's cluster.

5. The final case is the same as case 4, except that p does not have any tasks in its task pool (lines 20–31). In this case, p first checks its approximation of the load of PEs from its cluster. If this approximation tells it that at least one PE with non-zero load exists in the cluster, the message is forwarded to the cluster's most heavily loaded node. Otherwise, If no PE with non-zero load exists in p's cluster, the message needs to be forwarded to the head PE of some other cluster. In this case, the message age is increased, and if it has reached the limit (`MAX_VISITED_CLUSTERS`), it is sent back to the head PE of the thief. Alternatively, if the message has not reached its age limit, p checks its approximation of the load of remote clusters, and chooses the one to which to forward the message via call to the `choosePE()` function. Finally, the message is forwarded to the head PE of the chosen cluster.

Algorithm 19 shows the function that is used to choose a destination PE to which to send (or forward) the steal message. It differentiates between local-steal and remote-steal messages by accepting the `locality` parameter. In the case of the local-steal,

the destination is chosen randomly. In the case of the remote-steal, it must be that the PE that has invoked the `choosePE` function is the head PE of some cluster (since if the non-head PE of a cluster receives the remote-steal message, it will either forward it to the thief, or to its head PE, but will never have to invoke the function for choosing a PE). In this case, the head PE checks its approximation of the load of other clusters, and splits the set of all remote clusters into three groups: ones with the load higher than the load of a thief's cluster; ones with lower, but on-zero load; and ones with zero load. If the first group is non-empty, a random cluster is chosen from it (with

Algorithm 18 Feudal Work Stealing – processRemoteStealMessage(PE pe,StealMessage message) function for head nodes

```
1: updateLoadInfo (pe,message)
2: if message.originatingPE == pe then
3:    if message.task != NULL then
4:       add message.task to pe's task pool
5:    end if
6:    remoteStealing=false
7: else if message.age == MAX_VISITED_CLUSTERS then
8:    send message to message.originatingPE
9: else if message.task != NULL then
10:    if message.originatingCluster == pe.cluster then
11:       send message to message.originatingPE
12:    else
13:       send message to message.originatingCluster head node
14:    end if
15: else if pe has tasks in its task pool then
16:    taskToSend = choose a task from pe's task pool
17:    message.task = taskToSend
18:    send message to message.originatingCluster head node
19: else
20:    if node with non-zero load in pe's cluster exists then
21:       nextTarget = node with highest load in pe's cluster
22:    else
23:       message.age++;
24:       if message.age < MAX_VISITED_CLUSTERS then
25:          nextTarget = choosePE (REMOTE)
26:       else
27:          nextTarget = message.originatingCluster head node
28:       end if
29:    end if
30:    send message to nextTarget
31: end if
```

probabilities of clusters being chosen proportional to their loads, so the clusters with higher loads are more likely to be chosen). If the first group is empty, a random cluster from the second group is chosen (following the same principle for selection based on their load as in the previous case). Finally, if the first two groups are empty, a random cluster from the third group is chosen.

Algorithm 19 Feudal Work Stealing – choosePE(Locality locality, PE p, int originatingLoad) function

```
1: if locality == LOCAL then
2:    target = choose a random PE from p's cluster
3: else
4:    noHighLoadTargets = 0
5:    noLoadTargets = 0
6:    for all clusters c such that c != p.cluster do
7:       if p.loadInfo[cluster].load > originatingLoad then
8:          noHighLoadTargets++
9:          add c to highLoadTargets
10:      else if p.loadInfo[cluster].load > 0 then
11:         noLoadTargets++
12:         add c to loadTargets
13:      end if
14:   end for
15:   if noHighLoadTargets > 0 then
16:      targetCluster = choose one of the clusters from highLoadTargets
17:   else if noLoadTargets > 0 then
18:      targetCluster = choose one of the cluster from loadTargets
19:   else
20:      targertCluster = choose a random cluster
21:   end if
22:   targetNode = targetCluster's head PE
23:   return targetNode
24: end if
```

Finally, Algorithm 20 shows how the synchronisation of the approximations of loads is done between the one attached to the remote-steal message (`message.loadInfo`), and the one which a head PE P has. (`p.loadInfo`).

5.4 The Evaluation of Feudal Stealing

In this section, we evaluate the performance of Feudal Stealing for the applications and computing environments we considered in Section 5.2. Our main goal is to discover whether the applications' speedups under Feudal Stealing relate to the speedups under

CRS and Hierarchical Stealing, for which we saw that they are the best algorithms of those that do not use any load information. Additionally, we want to compare Feudal Stealing with Perfect CRS (in order to investigate the difference in speedups when perfect and approximate load information are used in the same base work-stealing algorithm) and with Grid-GUM Stealing (in order to compare our approach of obtaining the approximation of system load to the fully-distributed one used in Grid-GUM).

Regarding the Grid-GUM stealing algorithm, a big difference between it and the other algorithms we consider is that in Grid-GUM a victim transfers more than one task to a thief in a single steal operation. This is not a usual approach in work-stealing on distributed environments, since the steal messages can get prohibitively large, which can have very negative impact on the environments with limited bandwidth. Additionally, in the applications where some of the tasks generate additional parallelism (the kind of applications we consider in this chapter), transferring more than one task in a single steal operation can result in too much work being transferred to the thief and in overall load imbalance. In fact, we will see that for only one application (DC-FixedPar(100,25,4)) does Grid-GUM give a better speedup than Perfect CRS, which transfers only one task in each steal operation.

Our main focus in this section is on the applications we already considered in Section 5.2, for which there was a notable difference between the speedups obtained under the CRS and the Perfect CRS algorithms. For these applications the use of load information seems very important, and we want to test how good job of approximating the PE loads does Feudal Stealing do. Therefore, we will pay special attention to the DCFixedPar(40,k,4) and DCFixedPar(100,k,4) applications. However, in order to do a 'sanity check' and make sure that Feudal Stealing does not notably degrade the performance of the applications where the use of load information does not seem important, we will start with investigating the SimpleDC applications.

Algorithm 20 Feudal Work Stealing – updateLoadInfo(PE pe, StealMessage message) function

1: **for all** clusters **do**
2:    **if** message.loadInfo[cluster].timeStamp $<$ p.loadInfo[cluster].timeStamp **then**
3:       message.loadInfo[cluster].timeStamp = p.loadInfo[cluster].timestamp
4:       message.loadInfo[cluster].load = p.loadInfo[cluster].load
5:    **else**
6:       p.loadInfo[cluster].timeStamp = message.loadInfo[cluster].timestamp
7:       p.loadInfo[cluster].load = message.loadInfo[cluster].load
8:    **end if**
9: **end for**

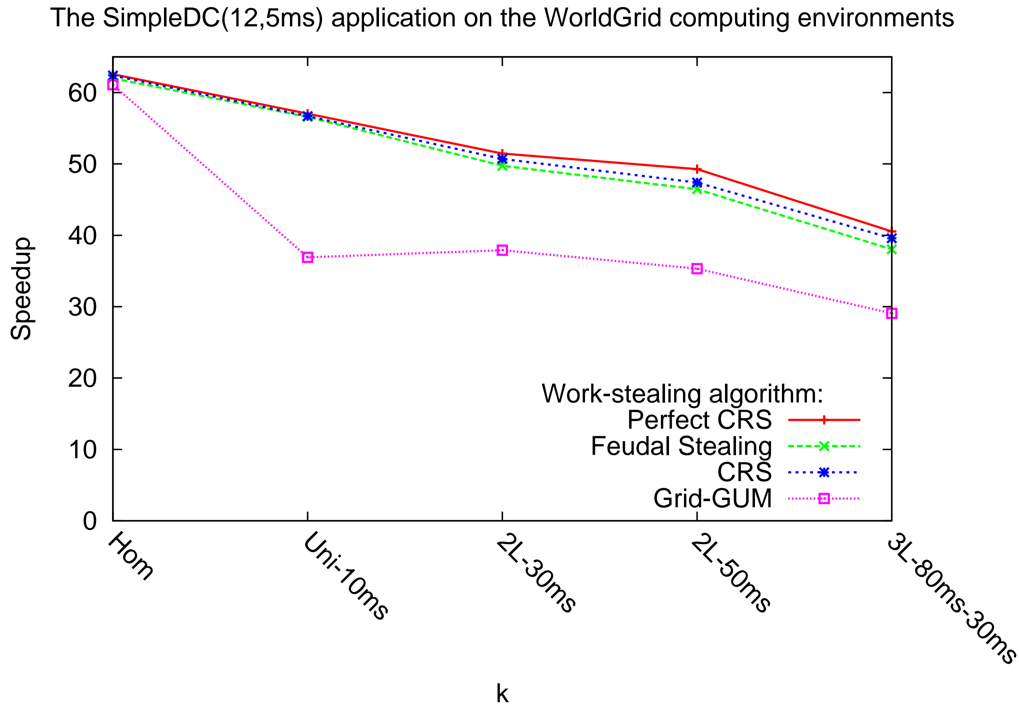


Figure 5.24: Speedups of SimpleDC(12,5ms) application on WorldGrid computing environments

5.4.1 The SimpleDC Applications

In our first experiment, we consider the SimpleDC(12,5ms) and SimpleDC(10,5ms) applications on the WorldGrid computing environments.

Figure 5.24 shows the speedups of the SimpleDC(12,5ms) applications under the work-stealing algorithms we are interested in. We have omitted Hierarchical Stealing from the comparison of speedups, as we have seen in Section 5.2.1 that it constantly performs worse than CRS. We can observe that Feudal Stealing gives a very similar speedups to Perfect CRS and CRS. Interestingly, the speedups under Feudal Stealing are slightly worse on more heterogeneous environments that those of CRS. The differences are, however, only minor, so there is no significant penalty in using Feudal Stealing. We can also observe that all variants of the CRS algorithm are significantly better than Grid-GUM.

Figure 5.25 shows the speedups of the SimpleDC(10,5ms) applications. For these applications, in Section 5.2.1 we have observed more notable improvements in the speedups of the Perfect CRS over the CRS algorithm. On Figure 5.25 we can see that the speedups under the Feudal and the CRS algorithms are very similar, and that

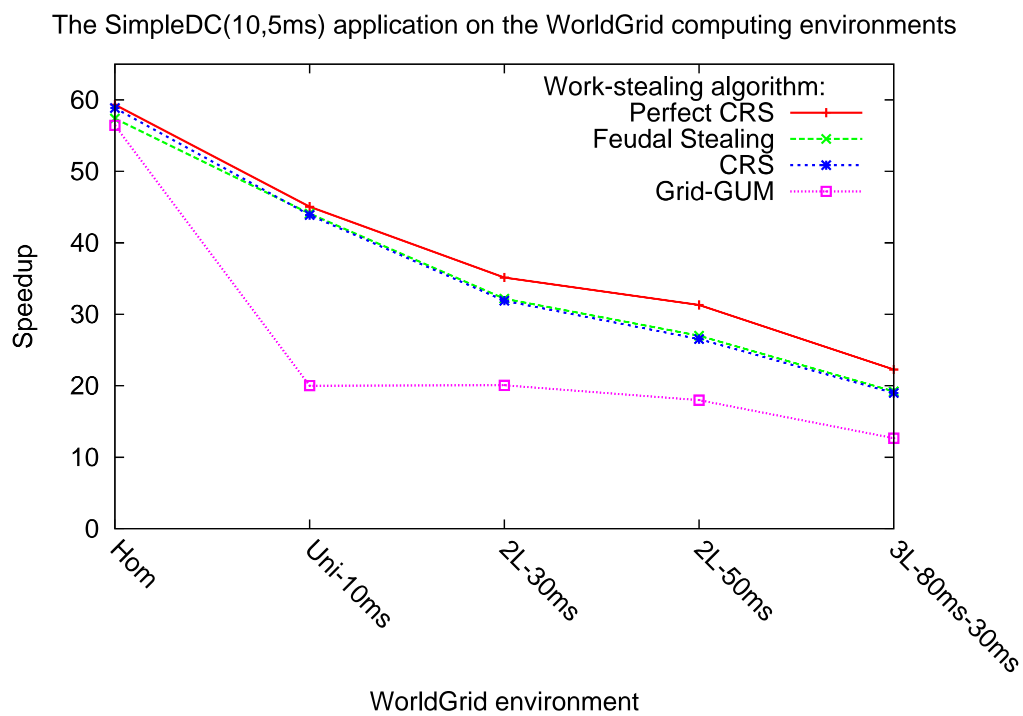


Figure 5.25: Speedups of the SimpleDC(10,5ms) application on the WorldGrid computing environments

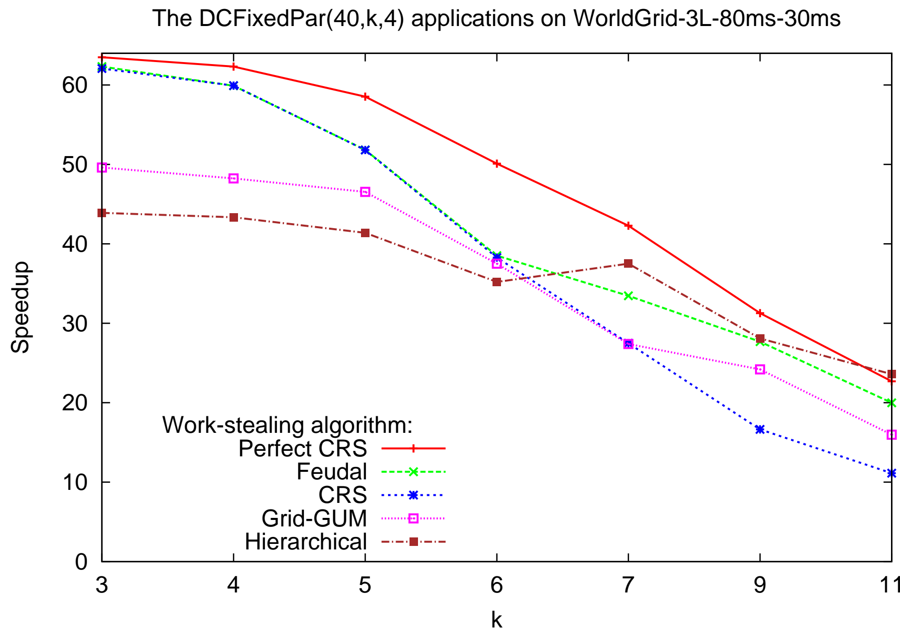


Figure 5.26: Speedups of the DCFixedPar(40,k,4) applications on the WorldGrid-3L-80ms-30ms environment

Feudal Stealing is only very slightly better. Again, similarly to Figure 5.24, we can see that the Grid-GUM stealing performs notably worse than all variants of CRS stealing.

From the experiments with the SimpleDC applications, we can conclude that Feudal stealing does not bring notable improvements to the speedups of applications, compared to the CRS stealing, if the applications are very regular and have a lot of parallelism. However, it also does not decrease the performance of these applications notably, so it is as good choice for them as is the CRS stealing, and certainly better choice than the Grid-GUM algorithm.

5.4.2 The DCFixedPar Applications

We now turn our attention to more irregular parallel applications, where nested-parallel tasks generate more sequential than nested-parallel tasks. As we have mentioned before, our special focus will be on the combinations of applications and computing environments for which the Perfect CRS gave notably better speedups than CRS in Section 5.2.

Figure 5.26 shows the speedups for the DCFixedPar(40,k,4) applications on the WorldGrid-3L-80ms-30ms environment, for $k \in \{3, 4, 5, 6, 7, 9, 11\}$. We have observed that for these applications, especially for higher values of k, there is a notable difference

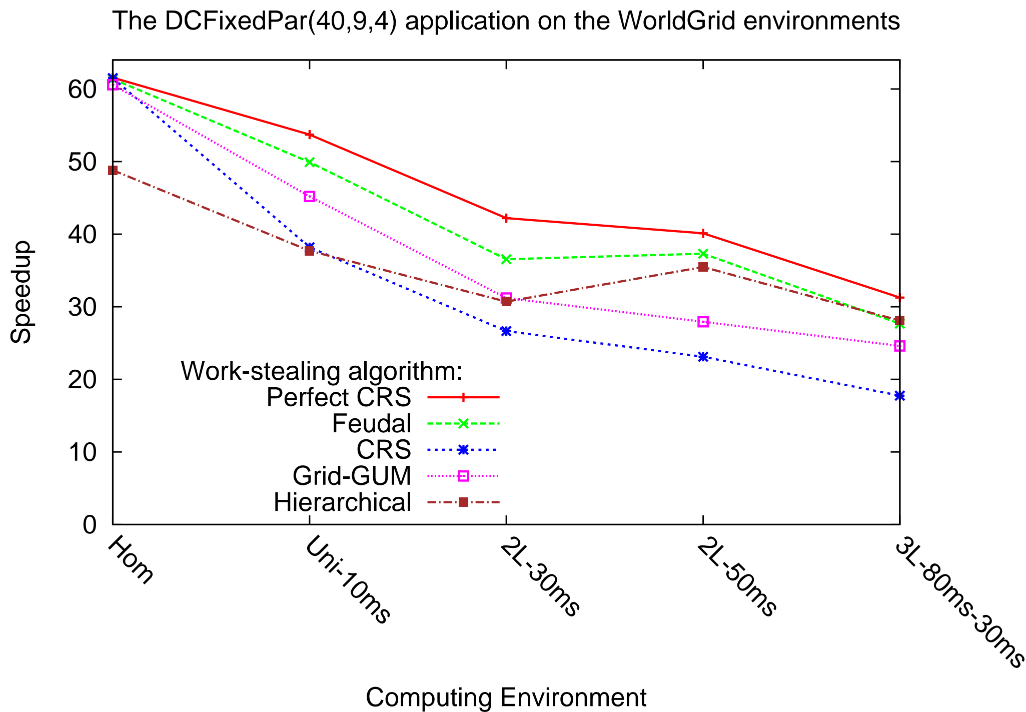


Figure 5.27: Speedups of the DCFixedPar(40,9,4) application on the WorldGrid environments

in speedups under the CRS and Perfect CRS algorithms. On this figure, we also include Hierarchical Stealing, as this algorithm showed to perform the best from all algorithms that do not use load information for highly-irregular applications. We can observe that for more regular applications (smaller values of k), the CRS and Feudal algorithms give almost identical speedups, and that both are significantly better than Hierarchical Stealing. Since there is a lot of parallelism in these applications, and there are a lot of possible steal victims, the use of load information with Feudal Stealing does not bring notable improvement (in fact, it even decreases the speedups slightly). However, for more irregular applications (for $k = 7, 9, 11$), we can observe that Feudal Stealing notably outperforms CRS, and that the applications' speedups get very close to the ones obtained under the Perfect CRS algorithm. Feudal Stealing does not, however, outperform Hierarchical Stealing for these applications, but the speedups under these two algorithms are very similar. We can also observe that the Grid-GUM stealing is better than CRS for more irregular applications, but is constantly worse than both Perfect ACRS and Feudal Stealing.

From Figure 5.26, we have observed that the more irregular the application is, more difference there is in speedups under the Feudal and CRS algorithms (in favour

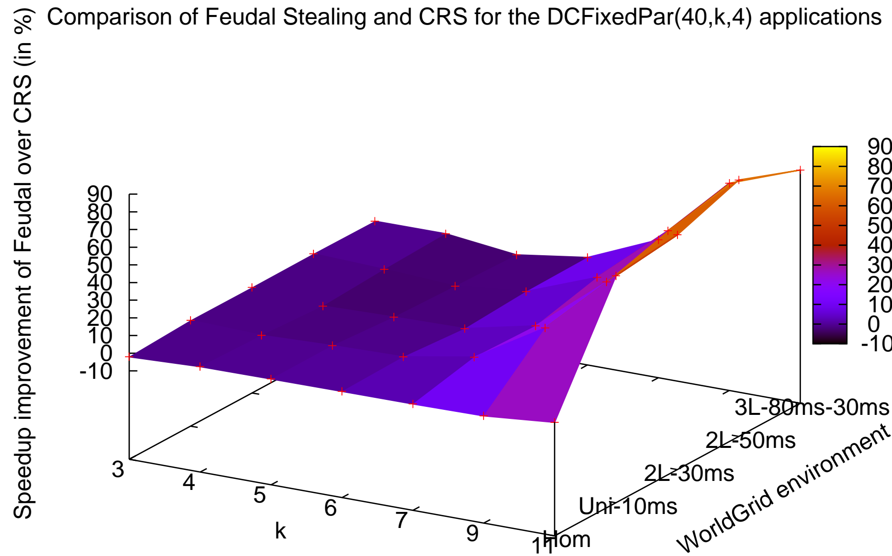


Figure 5.28: Improvements of Feudal Stealing over CRS for the DCFixedPar(40,k,4) applications on the WorldGrid environments

of Feudal Stealing). A similar trend can be observed if we consider the heterogeneity of underlying computing environment for fixed (irregular) applications. That is, for a fixed application, the more heterogeneous the computing environment on which it is executed is, the more improvements in speedup does the Feudal stealing bring to CRS. For example, Figure 5.27 shows the speedups of the DCFixedPar(40,9,4) application on the WorldGrid environments. We can again observe that Feudal Stealing performs better than CRS, and the difference in the performance between the two algorithms (in terms of the percentage of speedup improvement that Feudal Stealing brings to CRS) gets bigger the more heterogeneous the computing environment is. We can also observe that Feudal Stealing outperforms Hierarchical Stealing for all environemnts, except for WorldGrid-3L-80ms-30ms. The Grid-GUM stealing is constantly better than CRS, but is notably worse that both Feudal Stealing and Perfect CRS.

Figure 5.28 shows the speedup improvements of Feudal Stealing over CRS for all of the DCFixedPar(40,k,4) applications on all of the WorldGrid environments we considered. We can observe that, for more regular applications, the improvements are very small. In many cases, CRS even gives slightly better speedups. However, for highly irregular applications (k being 7, 9 or 11), we can observe a sharp increase in the improvements, especially on more heterogeneous environments. For example, for

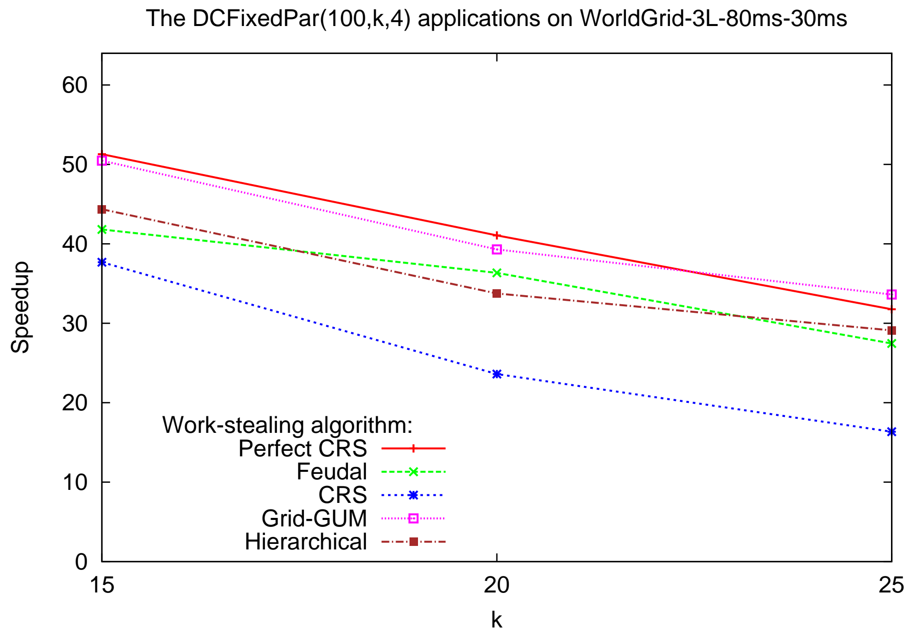


Figure 5.29: Speedups of the DCFixedPar(100,k,4) applications on the WorldGrid-3L-80ms-30ms environments

DCFixedPar(40,11,4) applications on the WorldGrid-3L-80ms-30ms environment, we can observe the speedup improvement of 90%.

Finally, we consider the DCFixedPar(100,k,4) applications, for $k \in \{15, 20, 25\}$. As a reminder, these applications are highly irregular, they have many more sequential than nested-parallel tasks and a very large number of tasks.

Figure 5.29 shows the speedups of the considered algorithms on the WorldGrid-3L-80ms-30ms computing environment. Interestingly, we can observe here that for $k = 25$ the Grid-GUM stealing gives a better speedup that Perfect CRS. This application is the example of the situation where sending more than one task in a single steal operation brings very good benefits to the load balancing. Nested-parallel tasks in this application create a lot of subtasks (100), but most of these are sequential (96 sequential, as opposed to 4 nested-parallel). Therefore, a PE that executes the nested-parallel task will have a large number sequential tasks in its task pool. A thief that steals from this PE will, under the Grid-GUM stealing, steal half of its tasks. Since most of the stolen tasks are sequential, the thief will not steal too much work and the benefit of stealing multiple tasks is that the PEs close to this will be able to obtain work from it. This does not happen under Feudal Stealing or Perfect CRS, where the thief would steal just a single (probably sequential) task.

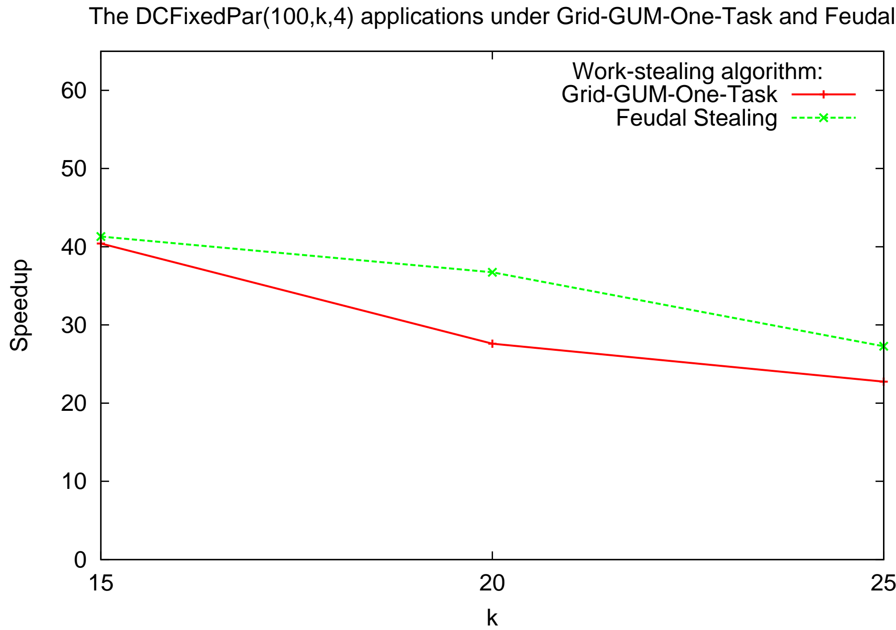


Figure 5.30: Speedups of the DCFixedPar(100,k,4) applications on the WorldGrid-3L-80ms-30ms environments under Grid-GUM-One-Task (the version of Grid-GUM stealing where only one task is transferred in one steal operation) and Feudal Stealing.

To see that the number of tasks sent on a single steal operation is really the reason for better performance of the Grid-GUM stealing, compared to Feudal Stealing, Figure 5.30 compares the speedups of the DCFixedPar(100,k,4) applications on the WorldGrid-3L-80ms-30ms environment under the Grid-GUM-One-Task stealing and Feudal Stealing. The Grid-GUM-One-Task stealing is identical to the Grid-GUM stealing, except that only one task is sent in one steal operation. We can clearly see that Feudal Stealing outperforms this version of the Grid-GUM stealing. From this we can conclude that the decisive factor for better performance of the pure Grid-GUM stealing on Figure 5.29 indeed is sending more than one task in one steal operation.

As for the comparison between Feudal Stealing, CRS and Hierarchical Stealing on Figure 5.29, we can again observe that the Feudal stealing gives much better speedups than CRS, and that the speedups under it are very close to the ones under Perfect CRS. We can also observe that Feudal Stealing and Hierarchical Stealing perform about the same, with Hierarchical Stealing being slightly better for $k = 25$ (for the application with the highest degree of irregularity).

Similarly to the experiments with the DCFixedPar(40,k,4) applications, we also want to observe how does the performance of Feudal Stealing changes, compared to Perfect CRS, CRS and Hierarchical Stealing, for a fixed application as the computing

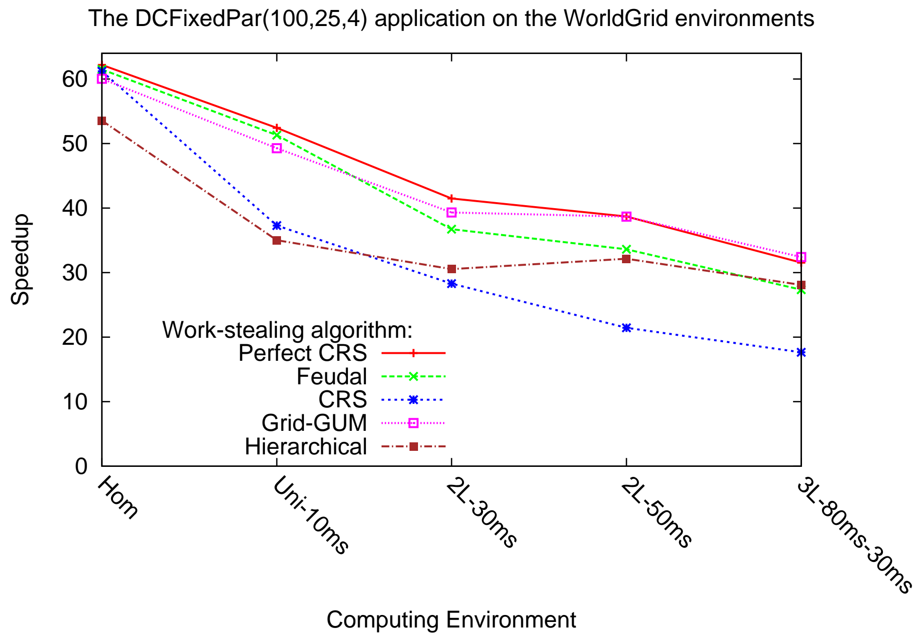


Figure 5.31: Speedups of the DCFixedPar(100,25,4) application on the WorldGrid environments

environment on which it is executed gets more heterogeneous. Figure 5.31 shows the speedups of the DCFixedPar(100,25,4) application on different WorldGrid computing environments. We can see similar results as in the case of the DCFixedPar(40,k,4) application. When the heterogeneity of the environment increases, the speedup difference between Feudal Stealing and the CRS increases in favour of Feudal Stealing. Also, Feudal Stealing outperforms Hierarchical Stealing on all environments, except for WorldGrid-3L-80ms-30ms.

Finally, Figure 5.32 shows the improvements of Feudal Stealing over CRS for all of the DCFixedPar(100,k,4) applications on all of the WorldGrid environments that we considered. We can observe that the improvements are increasing both as the application irregularity increases (when computing environment is fixed) and as the heterogeneity in the computing environment increases (for a fixed application). Therefore, the best speedup improvements (of 70%) can be observed for highly-irregular applications (e.g. DCFixedPar(100,25,4)) on very heterogeneous computing environments (e.g. WorldGrid-3L-80ms-30ms).

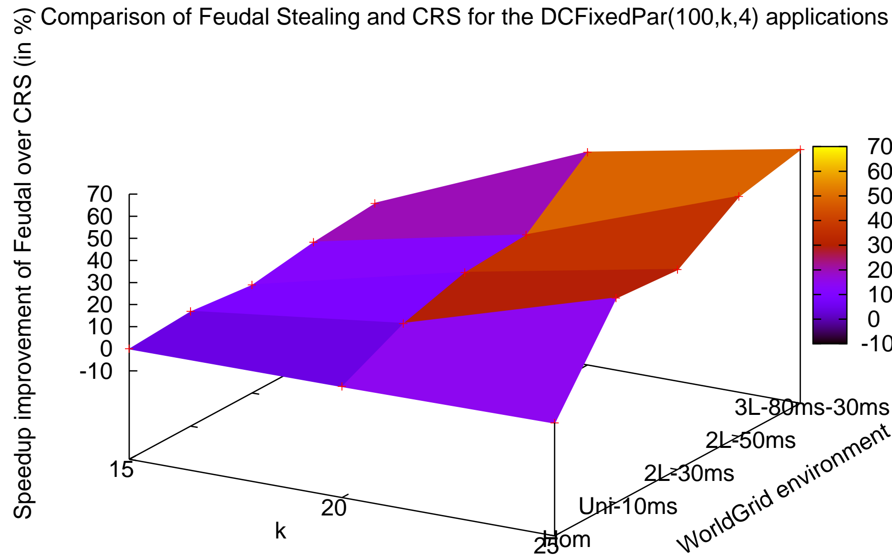


Figure 5.32: Improvements in speedups of Feudal Stealing over CRS for the DCFixedPar(100,k,5ms,0.1ms,0.1ms,4) applications on the WorldGrid environments

5.4.3 Why is Feudal Stealing Better than CRS and Grid-GUM?

The main motivation for developing the Feudal Stealing algorithm was to make better selection of targets when stealing is done over high-latency networks. Stealing within a cluster in Feudal Stealing is done in the same way as in CRS. Our main hypothesis was that the percentage of successful remote-steal attempts (i.e. those where a thief and a target are from different clusters) will be much higher under Feudal Stealing, and that this will be the reason for its better performance for applications where there is a small number of nested-parallel tasks, therefore, the thieves are unable to obtain work locally most of the time.

To see that our hypothesis holds, we have measured the percentage of successful remote-steal attempts under all of the algorithms we considered in this section for the DCFixedPar(40,k,4) and DCFixedPar(100,k,4) applications. Figures 5.33 and 5.34 give us the insight into this. As expected, the Perfect CRS algorithm has the highest percentage of successful remote steal attempts. The fact that this percentage is not 100 comes from the fact that, when no target has load higher than 0, a thief will attempt stealing from a random target, which, of course, might not have any work to send. The next best percentage is for Feudal Stealing. We can see that this percentage

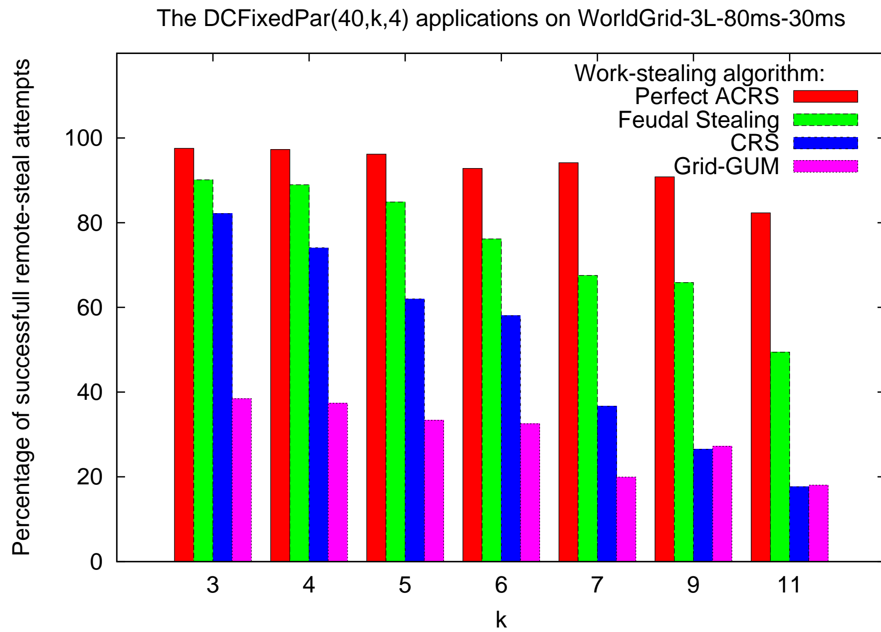


Figure 5.33: Percentage of the successful remote-steal attempts under Perfect CRS, Feudal Stealing, CRS and Grid-GUM for the DCFixedPar(40,k,4) applications on the WorldGrid-3L-80ms-30ms environment

is above 50% even for very irregular applications with very few nested-parallel tasks (e.g. DCFixedPar(40,11,4)). Additionally, we can see that Feudal Stealing makes many more successful remote-steal attempts than either CRS or Grid-GUM. We can also see on both figures that the difference in percentages of successful remote steal attempts between the Feudal stealing and CRS and Grid-GUM gets higher the more irregular the application is. Finally, we can also conclude that Feudal Stealing does a much better job of estimating the PE loads than Grid-GUM does, since many more remote-steal attempts are successful under it.

5.4.4 Summary

In this section we presented the evaluation of Feudal Stealing for irregular parallel applications on the WorldGrid computing environments (described in Figure 5.1, page 103). We have showed that Feudal Stealing gives better speedups for more irregular applications than either the CRS algorithm (on which it is based, and which does not use any load information) or the Grid-GUM stealing. For more regular applications, Feudal Stealing performs approximately the same as CRS, and only in a few cases does it give a slightly worse speedup. We can, therefore, conclude that it is worthwhile

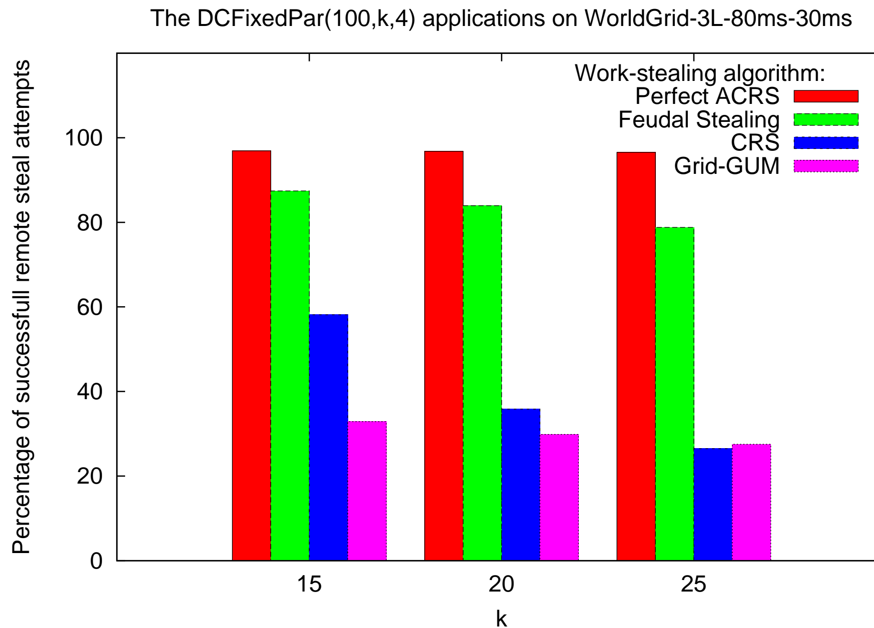


Figure 5.34: Percentage of successful remote steal attempts under the Perfect CRS, Feudal Stealing, CRS and Grid-GUM for the DCFixedPar(100,k,4) applications on the WorldGrid-3L-80ms-30ms environment

using Feudal Stealing instead of CRS – the improvements in speedups for more irregular parallel applications over CRS are very good (up to 90%), whereas the speedups for regular applications are very similar between the two algorithms.

Additionally, we showed that the speedup difference between Feudal Stealing and CRS is larger the more heterogeneous computing environments is. This shows that the heterogeneity of the computing environment also plays an important role in the decision in which of these two algorithms to use.

A comparison between Feudal Stealing and Hierarchical Stealing has shown that Feudal Stealing gives better speedups both for less irregular applications (on all environments) and for highly-irregular applications on less heterogeneous computing environments. The only cases where Hierarchical Stealing slightly outperforms Feudal Stealing are highly-irregular applications on highly-heterogeneous computing environments. However, even in these cases, the differences in speedups between the two algorithms are not big. Therefore, we can recommend using Feudal Stealing instead of Hierarchical Stealing, except in the cases where highly-irregular applications need to be executed on highly-heterogeneous environments, and where small improvements in speedups that Hierarchical Stealing can give are crucial.

Our main assumption was that the use of load information in Feudal Stealing

notably increases the chances of the thieves obtaining work from the remote targets in fewer attempts. We showed that the percentage of successful remote steal attempts under Feudal Stealing is much higher than under CRS or the Grid-GUM stealing. This shows that the Feudal stealing has exactly those benefits that we predicted – the ability to quickly obtain work over high latency networks, with few steal attempts.

5.5 Conclusions

In this chapter we attempted to answer the question of how to choose steal targets during the execution of irregular parallel applications. Our main assumption was that the use of dynamic information about PE loads can help in answering this question. We split the problem of using the load information to drive stealing decisions into two parts.

The first part deals with the issue of the most appropriate way of choosing steal targets, under the assumption that *perfect* (i.e. totally accurate) load information is present in the runtime system. We have investigated which algorithm for selecting the targets should thieves use in this settings. Additionally, we have investigated how much would the state-of-the-art work-stealing algorithms used in modern runtime systems (described in Section 2.3.1) benefit from the presence of perfect load information. Since these algorithms were previously evaluated only for very regular applications (i.e. simple divide-and-conquer ones) on more homogeneous computing environments, the questions of how would they perform for highly irregular applications on highly heterogeneous computing environments, and whether they could benefit from the load information was open. In order to address both of these issues, in Section 5.2 we performed a number of experiments using different regular and irregular parallel applications on different computing environments

Our conclusions in the first part of the chapter were the following:

- The best way to use load information is to do stealing locally (within a cluster) and remotely (outside of a cluster) in parallel, and to choose random steal targets with work. We named this algorithm for choosing the targets the Perfect CRS algorithm, since it is essentially the CRS algorithm with added perfect load information.

- Of all the algorithms that we considered, when no load information is used the CRS algorithm gives the best speedups for most of the irregular parallel applications. However, for highly irregular applications, Hierarchical Stealing

outperforms it. On the other hand, when load information is added to the algorithms that we considered, the CRS algorithm performs the best for all these applications on all computing environments.

- All of the state-of-the-art work-stealing algorithms were able to benefit from the use of load information. This is especially the case for highly irregular parallel applications, where we have noted a significant improvement in their speedups with the use of load information. The improvements were especially significant on highly heterogeneous computing environments. The only exception was the Hierarchical Stealing algorithm, where we did not observe a clear correlation between the amount of improvement obtained with the use of load information and the heterogeneity of the computing environment.

After demonstrating that load information can indeed be useful in work-stealing, in the second part of this chapter we focused on the way in which an accurate approximation of this information could be obtained during the application execution. In Section 5.3 we presented the Feudal Stealing algorithm. This algorithm uses the same principle as CRS for selecting steal targets. In addition, it uses a combination of locally-centralised (i.e. within the cluster) and remotely-distributed ways of exchanging information about PE loads. The main goal of the exchange of load information is for PEs to obtain a reasonable estimation of the load of other PEs during the application execution. We can see Feudal Stealing as an "approximation" of the Perfect CRS algorithm in realistic conditions, where perfect load information is not available.

In Section 5.4 we evaluated Feudal Stealing for the same set of applications considered in Section 5.2. We compared it with the Perfect CRS, CRS, Hierarchical Stealing and Grid-GUM algorithms. We showed that Feudal Stealing gives better speedups than either CRS or Grid-GUM, especially for highly irregular applications, for which we concluded in Section 5.2 that the use of load information is crucial. For these applications, the speedups under Feudal Stealing are close to those that were obtained under Perfect CRS. Furthermore, we showed that Feudal Stealing has a much higher percentage of successful remote (i.e. wide-area) steals than either CRS or Grid-GUM. This shows that its way of approximating the load information is better than the mechanism used in Grid-GUM. We also showed that Feudal Stealing gives speedups that are approximately the same as under Hierarchical Stealing for highly-irregular applications on highly-heterogeneous computing environments, and that are better than speedups under Hierarchical Stealing for all other applications and computing environments.

In the next chapter, we turn our attention to the question of how victims should respond to the steal requests from thieves.

Chapter 6

Granularity-Driven Work Stealing

In this chapter, we consider the question of *how* a victim should respond to a steal attempt from a thief, i.e. what task to send as a response. We assume that the run-time system has information about the sizes (as defined in Section 4.2.1) of all tasks of the application being executed. In Section 6.2, we propose several task selection policies that victims can use to choose the tasks they send to the thieves. In addition to information about the task sizes, these policies also rely on information about the network topology of the underlying computing environment. In Sections 6.3 and 6.4, we present the evaluation of these policies using both simulations and their implementation in Grid-GUM. We evaluate the extent to which the applications' speedups can be improved under the policies we propose, in comparison to the trivial FCFS task selection policy, which is used in most of the state-of-the-art work-stealing algorithms.

Most of the material in this chapter was published in Janjic and Hammond [JH10].

6.1 Introduction

In the previous chapter, we have considered the question of *where* thieves should look for work during the application execution. To ensure that thieves are not idle for a long time, it is important to select the stealing targets smartly. We have seen that the strategy to do remote and local stealing in parallel seems to be the best choice for a wide range of parallel applications, especially if PEs have a good approximation of the loads of the PEs in the computing environment. However, we have thus far ignored another very important question that arises during work-stealing – namely, if a victim has more than one task in its task pool, *what* task should it send as a response to the thief's steal attempt. Sending the "right" task to the thief can make that thief, and possibly neighbouring PEs as well, busy for a long time, thus eliminating the need for

them to look for work more often. This results in increased utilisation of thieves and, consequently, in the better speedups.

We proceed to investigate different policies for selecting the tasks to be sent as responses to the steal attempts. These policies also deal with selection of tasks that PEs should execute next (once the tasks currently being executed finish or are blocked). The situation where a PE has finished task execution and needs to choose the next task to execute constitute a special case of steal operation, where the PE steals a task from itself.

The most basic task selection policy is First-Come-First-Served (FCFS), where the victim always chooses the oldest task from its task pool, either for execution or to be sent to a thief. To consider any other policy only makes sense if the application's tasks are sufficiently "different" in some way, and if there is a difference in cost of sending different tasks to different PEs. In other words, the application needs to have a non-zero degree of irregularity, and the computing environment needs to be heterogeneous to some extent.

All the work-stealing algorithms proposed in literature assume that FCFS is the best task selection policy to use when the thief and the victim are different, and Last-Come-First-Served (LCFS) when the thief and the victim are the same. This approach seems natural for simple divide-and-conquer applications, as the data locality is preserved (since PEs execute the tasks most recently created tasks and whose data is still found in cache memory). The remote thieves in this case receive large chunks of work, as older tasks tend to be larger that the newer ones.

We can see that the FCFS-LCFS policy described above uses some implicit assumptions about the tasks that comprise the application being executed, such that, the older the task is, the larger it is, and the more parallelism it creates. While this is generally the case for the SimpleDC applications, in the case of the DCFixedPar applications, for example, we see that some of the older tasks might be sequential, while some of the newer tasks might have a lot of parallelism. For irregular SingleDataPar applications, task size is unrelated to age. Yet another drawback of the FCFS-LCFS policy is that it does not distinguish between thieves that are at different distances (in terms of communication latency) from the victim. The oldest task will be sent to the thief regardless of whether it belongs to the same cluster as the victim or not. This may not always be the best solution.

For our purposes, we assume that the runtime system has a good approximation of the size of each parallel task in the application. We propose novel policies that use this information to make better task selection decisions. The approximation of the

task size need not be perfect. We do not have to know the exact sizes of tasks, we only need to be able to compare two tasks, and determine which one is the larger.

Since the problem of selecting the task to execute or to send to the thief is orthogonal to the problem of selecting the stealing targets, the policies that we propose here are not specifically tied to any of the work-stealing algorithms considered in Chapter 5. In other words, our policies can be readily "plugged" into any of these algorithms. This makes the results and the conclusions that we obtain in this chapter very general and applicable to a wide class of runtime systems.

6.2 Granularity-Driven Task Selection Policies

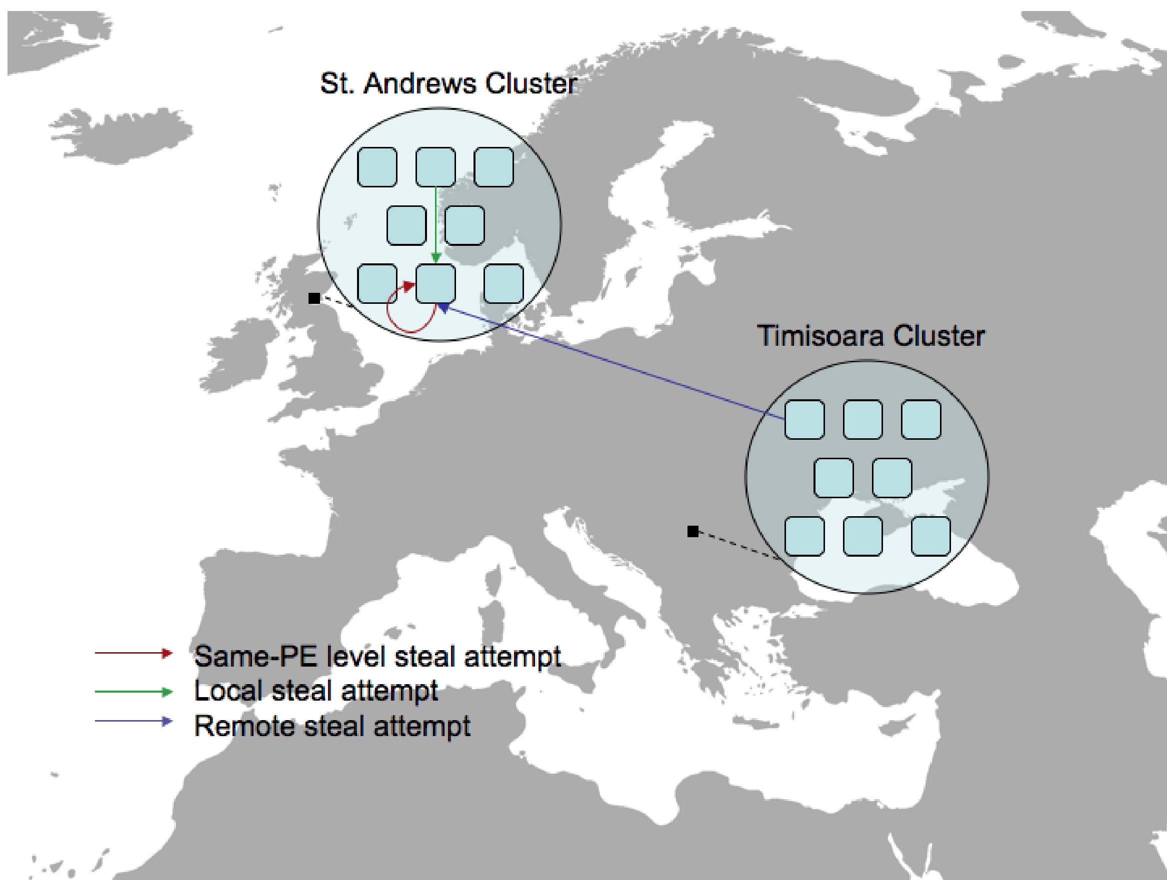


Figure 6.1: A computationally-uniform Grid

Since our focus is on computationally uniform distributed computing environments, the main problem for work-stealing that comes from the environment side is the presence of different communication latencies between a victim and different thieves. Consider the example computing environment on Figure 6.1, where a cluster that comprise

8 PEs, located in St. Andrews, is connected to the similar cluster in Timisoara, Romania. We can observe that a steal attempt can arrive to a victim from three different latency levels:

1. *same-PE level* – When a PE finishes executing a task (or when the executed task gets blocked on communication), it needs to choose one of the tasks from its task pool and execute it. As we have already mentioned before, we can see this as a special case of stealing, where a thief is attempting to steal from itself. The latency between the thief and the victim in this case is zero.

2. *same-cluster (local, LAN) level* - When a victim receives a steal attempt from a thief that is in the same cluster. We assume that in this case the latency between the thief and the victim is low, but that it is still higher than in the previous case.

3. *remote-cluster (remote, WAN) level* - When a victim receives a steal attempt from a thief from the remote cluster. In this case, we assume that the latency between the thief and the victim is much higher than in either of the previous two cases.

In this example, we distinguish between only two levels of communication latencies in the environment – one between PEs in the same cluster and one between PEs in different clusters. Of course, in the environments consisting of many clusters, the latency between various clusters can be highly different (take, for example, more heterogeneous WorldGrid environments we studied in Chapter 5). In addition, some clusters might consist of multicore machines[1], and the latency between different cores on the same machine is much lower than the latency between different machines. Therefore, both the same-cluster and remote-cluster levels can themselves consist of several different levels of latencies. However, most of the time it is enough to consider just two levels, since usually the latencies in the remote-cluster level are a few orders of magnitude higher that the latencies in the same-cluster level, and the differences between the latencies inside the individual levels are not too significant.

Consider what happens during the execution of parallel applications we consider in this thesis on distributed environments. At the beginning of the execution, the main task (possibly after some initial sequential computation) creates a set of child tasks and then blocks until all child tasks finish execution and (if they are offloaded) send their results back. These child tasks can, of course, create their own child tasks. If a

[1] In our setup, we consider each core of a multicore machine to be a separate PE

task is stolen from the PE that is executing its parent task, it needs to be transferred to the thief. In our simulated environment, each task can fit into a single message. In realistic runtime-systems, however, depending on the size of the data that needs to be transferred in order to execute the task, the task transfer may involve sending multiple messages from the victim to the thief. In the case of Grid-GUM, since data fetching is done lazily, the task transfer may involve the exchange of multiple `FETCH` and `RESUME` messages, which increases the overhead in task transfer. After the task finishes execution on the thief, its result needs to be sent back to the victim. Again, in realistic runtime systems, this may involve sending multiple messages, and in the case of Grid-GUM, the exchange of the series of `FETCH-SCHEDULE` messages is needed.

From the discussion above, we can see that executing a task on some PE, other than the one where its parent task is, involves possibly large overheads. However, in many cases these overheads are constant, no matter how large (in terms of the task size, as defined in Section 4.2.1) the task is. Therefore, sending a large task from a victim to a thief has the same overhead as sending a small one. The question is whether we can use this fact, plus the knowledge about the sizes of application tasks to select different tasks at different latency levels in order to better hide the WAN latency (maybe at the expense of increasing the number of messages sent over LAN).

When the tasks sizes are not known (or approximated) in advance of the execution, we can use the same policy which is generally used for the divide-and-conquer applications – First-Come-First-Served (FCFS) policy at the same-cluster and remote-cluster levels, and Last-Come-First-Served (LCFS) at the same-PE level. The last task that a PE creates will be the first one to be executed on it, and the first task that the PE creates will be the first one to be sent to a thief. In the following discussion, we will refer to this policy simply as FCFS, and we will use it as a baseline against which we will compare more advanced policies.

Assuming, however, that we have *a-priori* information about the size of every task, we can instead organise the task pool into a priority list ordered by task sizes and then use different selection policies at different latency levels. We will consider the following *granularity-driven* task selection policies for the three different latency levels:

- **Small-Small-Large (SSL)**. When responding to a steal attempt from the same-PE or the same-cluster level, select the smallest task. At the remote-cluster level, select the largest one. The rationale for this policy is that a victim wants to reserve as many large tasks as possible for the remote thieves, to compensate for the high communication costs that they will incur. The victim also wants to keep the remote thieves busy for a longer period than the PEs from the same cluster, so that they request work less often. This means that they would need to send fewer messages over high-latency networks, in the case that they repeatedly need to steal from the same victim. In this way, we aim to both improve the CPU utilisation and to decrease the number of messages sent (at least, the ones sent over high-latency networks), compared to the basic FCFS policy.

- **Small-Large-Large (SLL)**. Choose the smallest task only for the same-PE level, and select the largest task for all other levels. The rationale for preferring this policy to SSL is that the only important thing when answering to a steal attempt is to send the large task, and that it does not matter whether the attempt comes from the same or different cluster. The goal here is only to avoid the overheads of offloading too small tasks, and not to save the largest ones for more remote PEs. Using this policy, the PE that creates parallel tasks should, therefore, execute all or most of the small tasks and only large tasks should ever be offloaded. This is the policy that is implicitly used in all algorithms considered in Section 2.3.1, as for the simple divide-and-conquer applications it is the same as the FCFS policy.

- **Large-Large-Large (LLL)**. Always choose the largest task. This is the greedy approach, where we try to get the largest task executed as soon as possible (i.e. as soon as there is an idle PE to execute it). Smaller tasks are left to be executed later.

- **Large-Large-Small (LLS)**. Choose the smallest task for the remote-cluster level, and the largest one for the same-PE and same-cluster levels. In this way, we hope that the PEs nearby the victim will execute all large tasks, and by the time they finish, the remote PEs will finish executing the small tasks and send their results back. This, perhaps counter-intuitive, policy is actually optimal when the number of tasks altogether is approximately the same as the number of PEs.

Note that we are *not* trying to achieve load balance in such a way that each PE will execute approximately the *same number* of tasks, since for irregular applications

executed on heterogeneous environments, this kind of balance would result in very poor runtimes. Rather, our goal in most policies is to send large amounts of work to the remote PEs, so that they will execute a smaller number of large tasks. In this way, we expect to achieve a better balance of *actual work* across the entire environment, and also to minimise the impact of the communication costs.

The main assumption for the granularity-driven task selection policies is that there exists the irregularity in the sizes of the tasks that comprise the application that is executed. Since we have introduced the precise definition of the degree of irregularity of parallel application in Section 4.2.1, we can precisely relate the amount of application irregularity to the improvement that the granularity-driven task selection policies bring to the FCFS one.

In the remainder of the chapter, we investigate (using both the SCALES simulator and the implementation in Grid-GUM) the extent to which the proposed granularity-driven task selection policies can improve the speedups of irregular applications on heterogeneous computing environments, when compared to the FCFS policy. We also investigate the correlation between some characteristics of an application (namely, the number of application tasks, their mean task size and the degree of irregularity of the application) and the improvements that these policies bring. This enables us to observe for which kind of applications is it worth using granularity-driven task selection policies, and what are the limits on improvements in speedups that we can achieve.

6.3 Simulations Experiments

6.3.1 Overview

The purpose of the simulation experiments was to evaluate the extent of the improvements in speedups that the granularity-driven task selection policies bring to a wide range of irregular parallel applications. This shows us whether the knowledge of the sizes of tasks that the application comprise can play an important role in its load balancing on heterogeneous computing environments.

In most of the experiments, our focus will be on the SingleDataPar(t,m,d) applications (see Section 4.2). As a reminder, these applications comprise a master task, which immediately generates a number of sequential child tasks (with random sizes under the normal distribution, with the mean task size m and the standard deviation d), and then blocks until all of the child tasks complete their execution. After all of the child tasks send their results back to the master task, the master task finishes. Since in this case all of the parallelism is created on one PE, we can assume that the

thieves always tries to steal from this PE (in the rest of the discussion, we will call this PE the *main PE*). We, therefore, do not need to pay attention to *where* the thieves look for work. Wrong decisions of this kind can interfere with the results we are trying to obtain, since the poor speedup of the application can also come from the wrong selection of steal targets.

Note that, for fixed values of t,m and d, SingleDataPar(t,m,d) represents a class of applications, rather than a single application (as was the case, for example, for DCFixedPar(n,k,C_{seq},t), which represented a unique application). Different applications that belong to the same SingleDataPar(t,m,d) class may have tasks of different sizes, but they all share the same mean task size, standard deviation and the number of tasks. This further means that if we execute two different applications that belong to this class, we will likely get different speedups. Therefore, for each experiment where we consider the speedup for a particular class of SingleDataPar applications, we have taken the average speedup of 100 different applications that belong to that class.

For the completeness of the discussion, we also consider some examples of the applications with nested parallelism (the DCFixedPar applications). For this kind of applications, we will assume that the underlying work-stealing algorithm used is Cluster-Aware Random Stealing with the perfect knowledge of dynamic PE loads (Perfect CRS, described in Section 5.1.1). As we have seen in Chapter 5, this algorithms makes the best decisions about where to send the steal attempts of all the ones we considered, so it makes sense to use it as a base for the granularity-driven task selection policies for applications with nested parallelism. Note that here, for fixed values of n, k C_{seq} and t, the DCFixedPar(n,k,C_{seq},t) is the unique application. However, task distribution under Perfect CRS work-stealing algorithm can be different in different executions of the same application, due to some degree of randomness. We have, therefore, similarly as for the SimpleDataPar application, in each experiment taken an average speedup over 100 exeucutions of the same application.

For the SingleDataPar(t,m,d) applications, we will consider each of the three application parameters (as a reminder, t is the number of application tasks, m is the mean task size, and d is the degree of irregularity) separately, and observe what impact different values of each of them have on applications' speedups under the granularity-driven task selection policies. Our aim is, therefore, not only to investigate whether the granularity-driven task selection policies can improve speedups of parallel application, but also for *what* applications are improvements the best/worst and also what policy is the best for what kind of applications

Concerning the computing environments that we study, in most of the experi-

ments we consider the environments consisting of just two clusters connected through a very high-latency network, since these kind of architectures (having two different latency levels) correspond the best to the task selection policies we investigate. Different environments that we consider will, therefore, differ in the number of PEs they consist of and the inter-cluster and inside-cluster latencies. Similarly to the environments considered in Section 5.2, we will denote this kind of environments by Grid(2,n,LANLat,WANLat), where n is the number of PEs in each cluster, LANLat is the communication latency between PEs in the same clusters, and WANLat is the communication latency between clusters. Again, for the completeness of discussion, we will also consider how our approach scales to more heterogeneous environments, where more than two latency levels exist.

Concerning the actual execution of applications, the setup of the SCALES simulator assumes that the scheduling is preemptive – that is, as soon as a victim receives a steal attempt, it will stop the execution of its currently executed task (if any), respond to the steal attempt, and then continue the execution. If a child task is executed on the same PE as its parent, there will be no communication costs. If, instead, it is offloaded to a different PE, then we assume that it will be packed, together with all of the data it needs for its execution, into a single message. When the child task completes, we again assume that its result, which needs to be sent to the PE where its parent task resides, can fit into one message. We will, therefore, have a fixed communication cost of two messages for each task transferred between PEs. This accurately models the eager work-stealing mechanism that has been implemented in many systems (Cilk, Satin, Javelin 3 etc.).

In the following discussion, for the applications with a single level of parallelism executed on two-cluster environments, we will adopt the following terminology. The *main PE* will refer to the PE where the main task is executed, the *main cluster* to the cluster where the main PE is, and the *remote cluster* to the other cluster (See Figure 6.2). For the environments comprising more than two clusters, remote cluster will refer to any cluster other than the one to which the victim that is currently considered belongs.

We now list the hypotheses we wanted to test about the performance of granularity-driven task selection policies:

- *For parallel applications with a larger number of tasks, either the SSL or the SLL policy will bring the best speedup* - Intuitively, for the applications with a large number of tasks, the strategy to save the largest tasks for PEs in the remote cluster should prove to be better than a greedy strategy to execute the largest

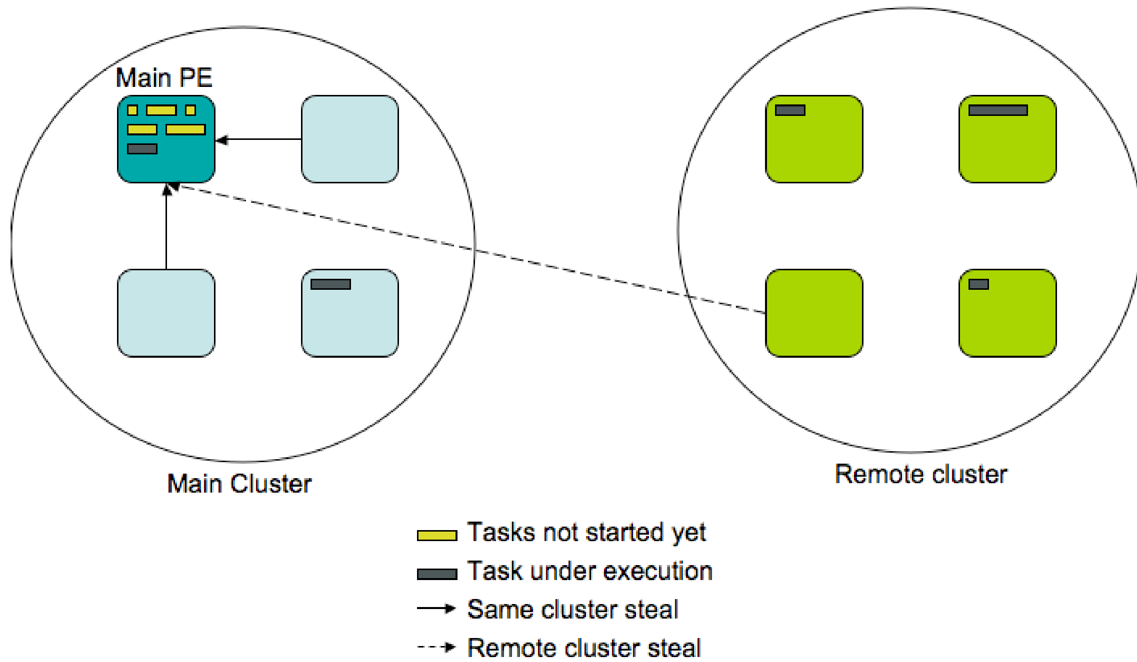


Figure 6.2: Snapshot of the execution of an application on the example environment that comprise two clusters, with 4 PEs in each cluster

task as soon as possible. In this way, the PEs from the remote cluster should spend less time in trying to obtain work, therefore increasing their utilisation. Consequently, we expected that the SSL or the SLL policy will bring the best speedups for applications that have enough parallel tasks. For smaller applications, LLL or LLS might as well be the policies to choose, as their greedy approach might be better in the short term.

- *Increases in speedups under the SSL and the SLL policy over the FCFS one come from better utilisation and a fewer number of steal attempts sent from the PEs from the remote cluster* - If, indeed, the SSL and the SLL policies show to be better than FCFS, then it will be due to a fact that the PEs are overall better utilised. For the applications with a single level of parallelism, this should come from the better utilisation of the PEs from the remote cluster, since larger tasks are executed on them, so their ratio between the "useful time" (the time spent in task execution) and the "idle time" (the time spent in looking for work) should be significantly better than under the FCFS policy. On the other hand, the utilisation of the PEs from the main cluster can drop under the SSL policy

(compared to FCFS), as smaller tasks get to be executed there. However, since the communication latency between the PEs from the main cluster is low, they are able to obtain work quickly, so, except for the applications with extremely fine grained tasks, we do not expect that the drop in utilisation will be significant. In other words, the increase of utilisation of the PE from the remote cluster should be far greater than the decrease of utilisation of the PEs from the main cluster.

- *The granularity-driven task selection policies will have more impact on the execution of applications with finer grained tasks* - We assumed that if the application's tasks are sufficiently coarse-grained, it becomes unimportant what task is chosen for offloading at what latency level. In this case, even with very high latencies, the cost of transferring a task over the network (and sending its result back) becomes negligible, as it is significantly lower that the cost of task's sequential part. On the other hand, for the applications with finer grained tasks, if the task granularity is not taken into account when decisions are taken about what tasks to send to the thieves, bad cases can happen where very fine grained tasks are transferred over high latency network, and where the costs of their transfer are far greater than the cost of their sequential execution. Therefore, we assumed that the granularity-driven task selection policies (except for the LLS one) will prevent these bad cases from happening, therefore improving the speedups of applications with finer-grained tasks.

- *The more irregular parallel application is, the better improvements can granularity-driven task selection policies bring* - Applications with a lower degree of irregularity will have tasks of similar size, so we do not expect for the granularity-driven task selection policies to have a big impact on the speedups of this kind of applications. We only expect that the improvements will be notable if task sizes are sufficiently variable. Furthermore, the more different the sizes are, the bigger difference is between offloading the "right" and the "wrong" tasks. As a consequence, we expected that the improvements under the granularity-driven task selection policies will increase as applications' degree of irregularity increases.

6.3.2 Applications with Variable Mean Task Size

In our first set of experiments, we focus on investigating the performance of the granularity-driven task selection policies for highly irregular parallel applications that comprise a large number of tasks. We observe how do the speedups under all considered policies change as the mean task size of the application changes.

The results that we present in this section are not sensitive to the particular two-cluster environment we consider. We will, therefore, focus on the environment where each cluster consists of 8 PEs, the inter-cluster latency is 30ms, and the latency inside clusters is 0.1ms. Similarly to the WorldGrid-2L-30ms environment considered in the previous chapter, this environment simulates connecting two clusters from different countries from the same continent. Unless we specify otherwise, this will be the default environment we consider in all of the simulation experiments. That said, we also present the results for other two-cluster environments, and we will devote a separate section for investigating how do the granularity-driven task selection policies scale on the environments consisting of more than two clusters.

We assume a single level of parallelism in applications. For all tested applications, we fix the degree of irregularity to be 0.9. In most of the experiments, we will fix the number of tasks in an application to be 100 per PE (which makes up to 1600 tasks for our default environment). That is, most of the time we consider SingleDataPar(1600,m,0.9) applications, where m is variable. We will also consider some applications with larger number of tasks. However, we will see that the results we present here are not sensitive to the number of tasks that an application comprise, as long as there are *enough* tasks.

Figure 6.3 shows the speedups obtained under all of the granularity-driven task selection policies, plus the FCFS policy, for the SingleDataPar(1600,m,0.9) applications with various values of m (mean task size). From the figure, we can observe that the SSL, SLL and LLL policies bring improvements to applications' speedups, compared to when the FCFS policy is used. We can also observe that the SSL policy gives the best speedups and that the LLS one is constantly the worst. We can, therefore, conclude that for these applications, the SSL policy is the best one and that the information about the task sizes can improve the applications' speedups, compared to when no such information is present (under the FCFS policy).

Figure 6.4 shows the improvements in speedups that the best policy (SSL) brings over the FCFS one for the applications considered in Figure 6.3. We can see that the improvements are mostly notable (above 10%) for the applications with finer-grained tasks. This fact agrees with our hypothesis that the better improvements under the granularity-driven task selection policies can be obtained for the applications with finer-grained tasks. For the applications with coarse-grained tasks, the improvements become smaller, but are still measurable. This is expected, since for these applications, even the smallest tasks are still quite large, so the overheads in their offloading are not too notable, and therefore reserving the large tasks for the PEs from the remote

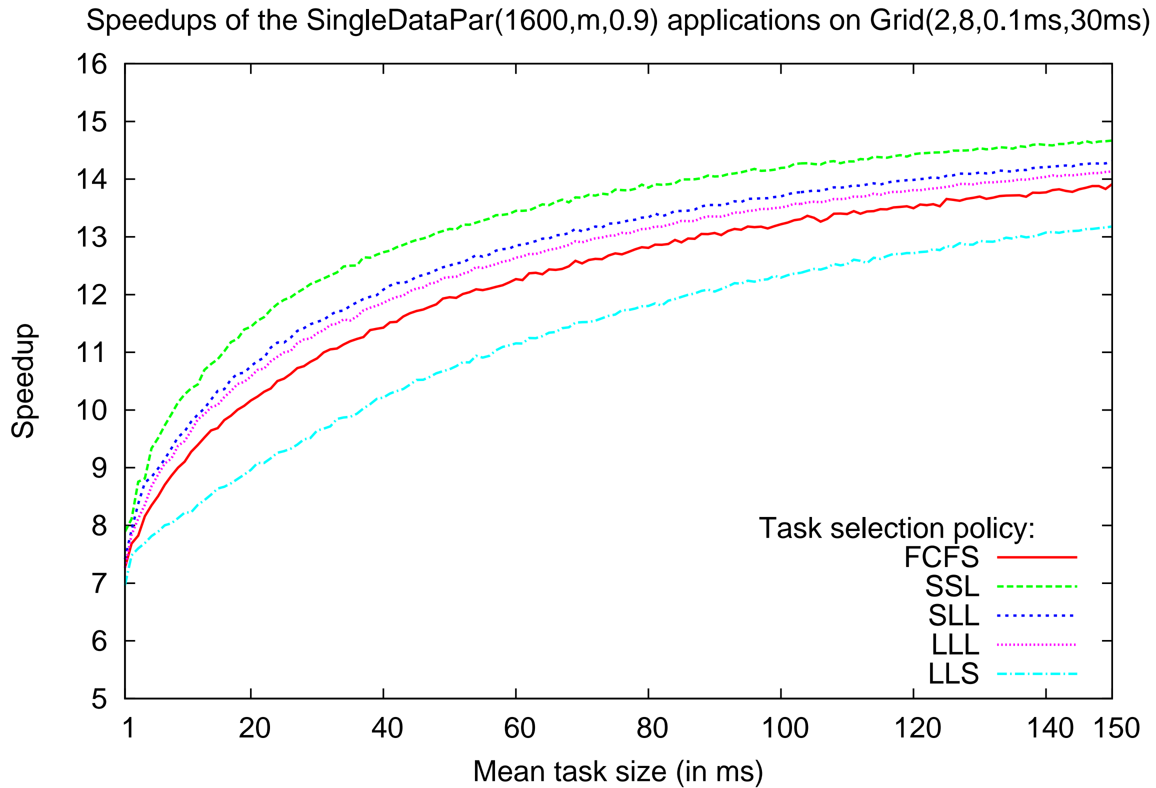


Figure 6.3: Speedups of the SingleDataPar(1600,m,0.9) applications on the Grid(2,8,0.1ms,30ms) computing environment

cluster does not make much difference for overall PE utilisation and the applications' speedups. Another thing that we can observe is that the SSL policy is *always* better than FCFS, which means for the applications considered here, the information about the task sizes is always useful at least to some extent.

From Figure 6.3, we can observe that the LLS policy performs the worst of all the policies that we consider, for all applications. This is somewhat expected, since for larger applications, it does not seem useful to save small tasks for the PEs from the remote cluster. Still, it is useful to consider the difference between speedups obtained under the best (SSL) and the worst (LLS) policy, for this is a kind of a best-case scenario for the use of information about the task sizes. This comparison shows us the difference in speedups obtained under the policies that make the best and the worst decisions about what tasks to send where. Some applications may create tasks whose sizes follow some particular pattern (where, for example, the tasks created earlier are smaller), so selecting the tasks under the FCFS policy may correspond to the LSS (the policy we do not consider here) or the LLS policy. Figure 6.5, therefore, shows the

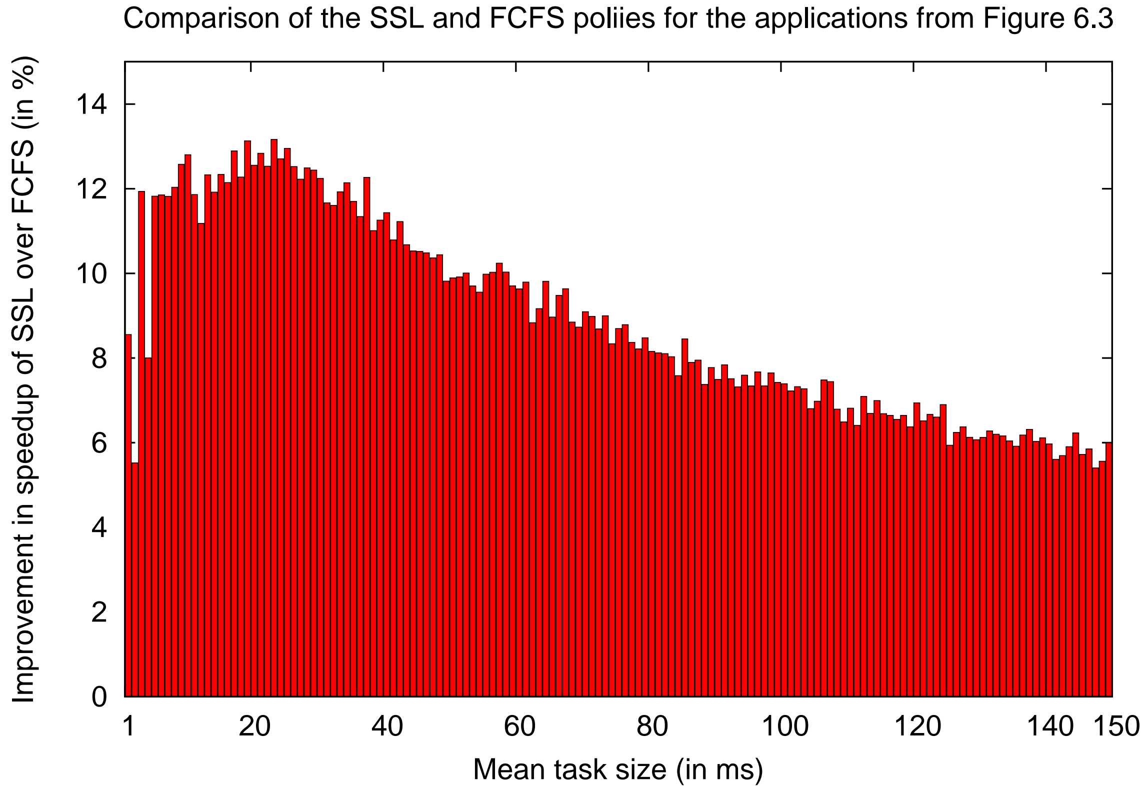


Figure 6.4: Improvements in speedups that the SSL policy brings over the FCFS one for the SingleDataPar(1600,m,0.9) applications

speedup improvements (in percentages) that the SSL policy brings over the LLS one for the applications considered on Figure 6.3. We can see that the improvements are between 20% and 30% for the applications with finer-grained tasks (with the mean task size of up to 80ms), and between 10 and 15% for the applications with coarse-grained tasks. This shows that we can obtain significant improvements in speedups if we use information about the task sizes in the right way.

Very similar results to the ones presented in Figures 6.3 – 6.5 hold for other computing environments which consist of two clusters that communicate over a high-latency network. The only difference is that the same speedups are obtained for the applications with larger mean task size in the case of environments with higher latency between clusters. Also, as we can expect, the applications' speedups are generally lower when the latency between clusters is higher. For example, Figure 6.6 shows the results equivalent to these on Figure 6.3 for the Grid(2,8,0.1ms,60ms) (Figure 6.6a) and Grid(2,8,0.1ms,90ms) (Figure 6.6b) environments. We can observe almost identical behaviour of the granularity-driven task selection policies as on Figure 6.3 – for

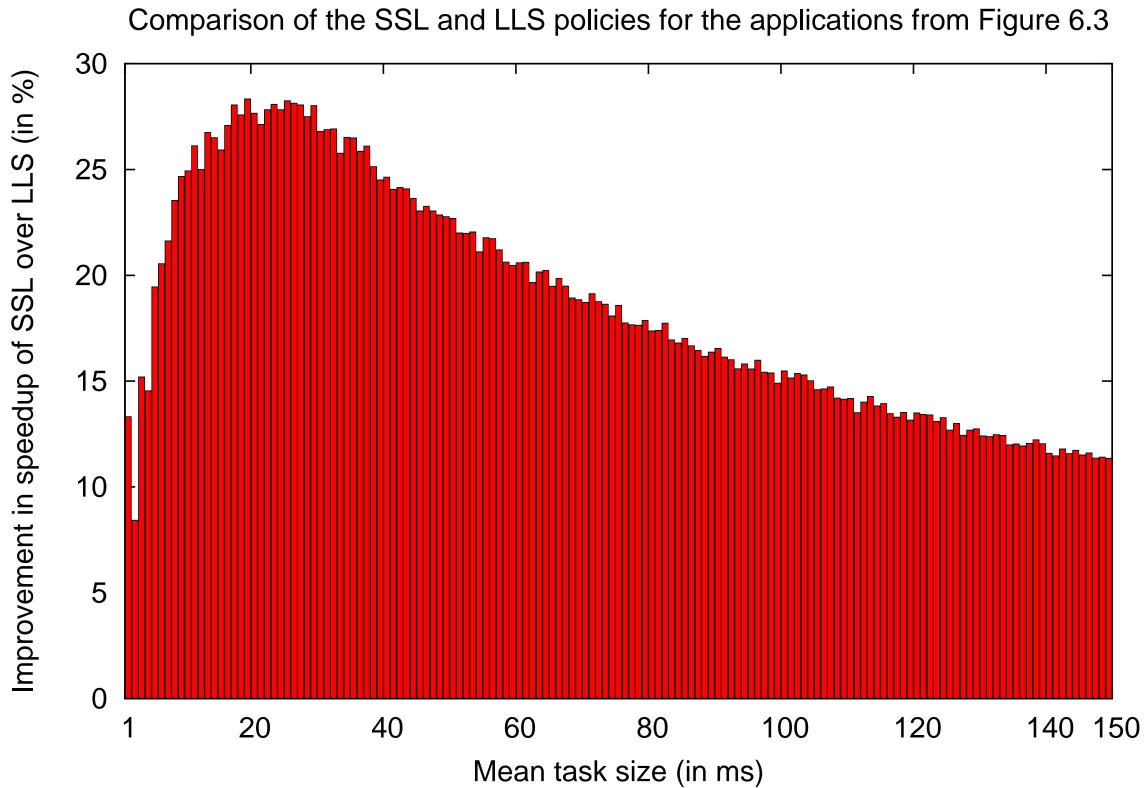
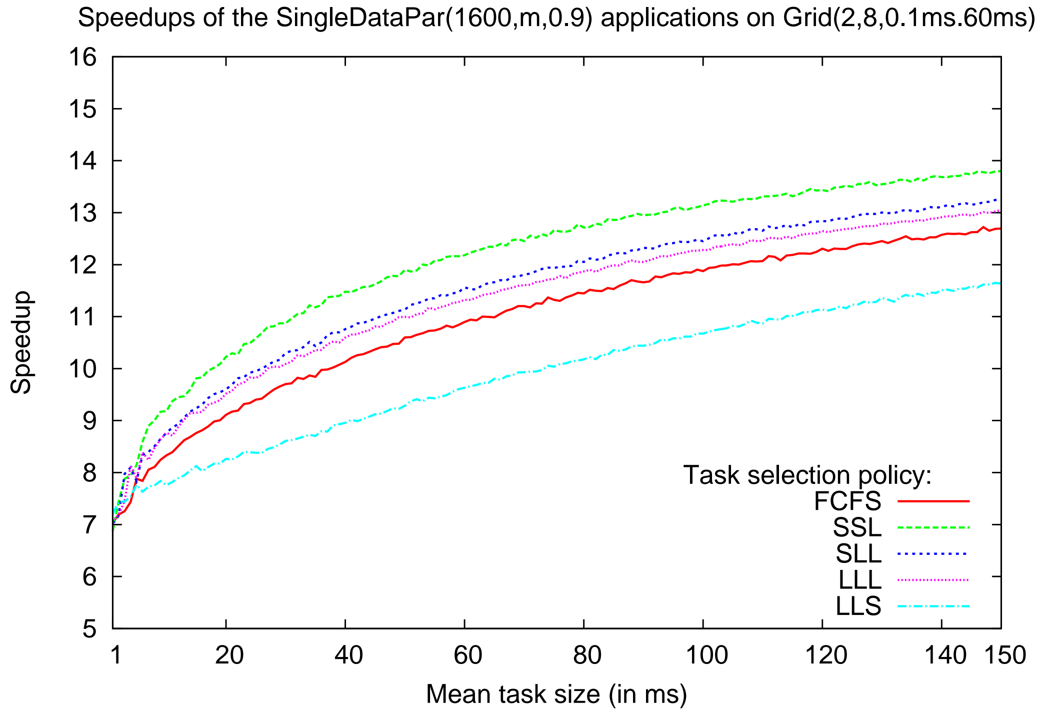


Figure 6.5: Improvements that the SSL policy brings over the LLS one for the SingleDataPar(1600,m,0.9) applications

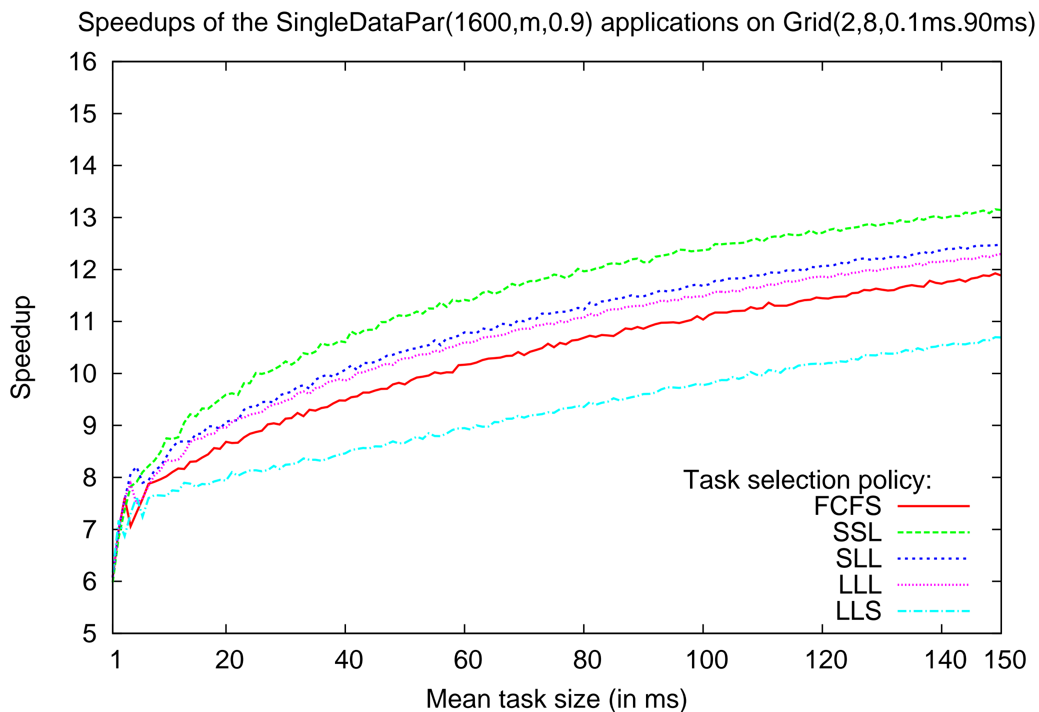
the applications with smaller mean task size, all of the considered policies give similar speedups, but as the mean task size of the applications increase, the SSL policy outperforms all other policies; the SSL, SLL and LLL policies outperform the FCFS one, and the LLS policy is constantly the worst one. We can, however, observe that the mean task size after which SSL starts to outperform FCFS (and other policies) gets slightly larger as the inter-cluster latency is increased.

Similarly, when we investigate the environments where the latency between clusters is the same as in our default environment (30ms), but which have higher latency inside clusters, we get more or less the same results. Figure 6.7 shows the results equivalent to these on Figure 6.3, but for the Grid(2,8,0.5ms,30ms) environment. Again, we can observe almost identical results as on Figure 6.3.

Finally, we briefly show how do the granularity-driven task selection policies perform on the environments with larger number of PEs. We will postpone more detailed analysis of the scalability of these policies to Section 6.3.5. Here, we show (Figure 6.8) the speedups of the same set of applications as on Figure 6.3 on yet another two-cluster



(a) Inter-cluster latency 60ms



(b) Inter-cluster latency 90ms

Figure 6.6: Speedups of the SingleDataPar(1600,m,0.9) applications on the Grid(2,8,0.1ms,60ms) and Grid(2,8,0.1ms,90ms) environments

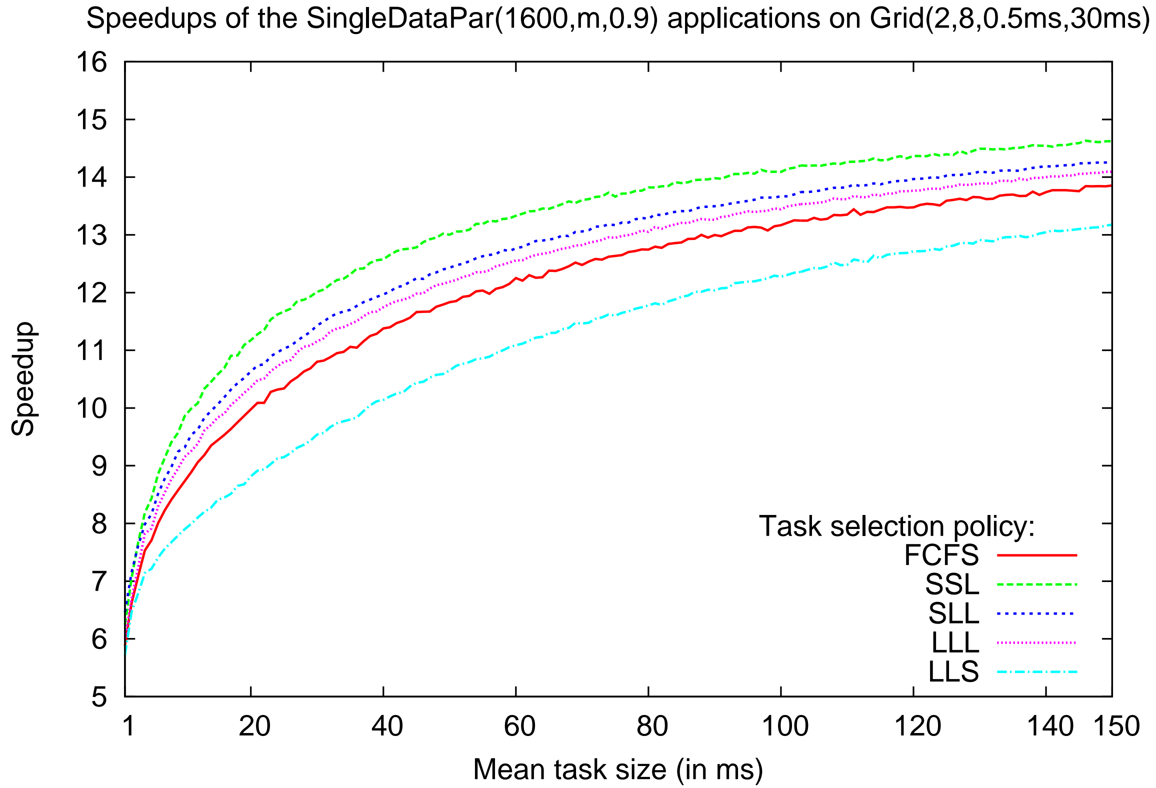
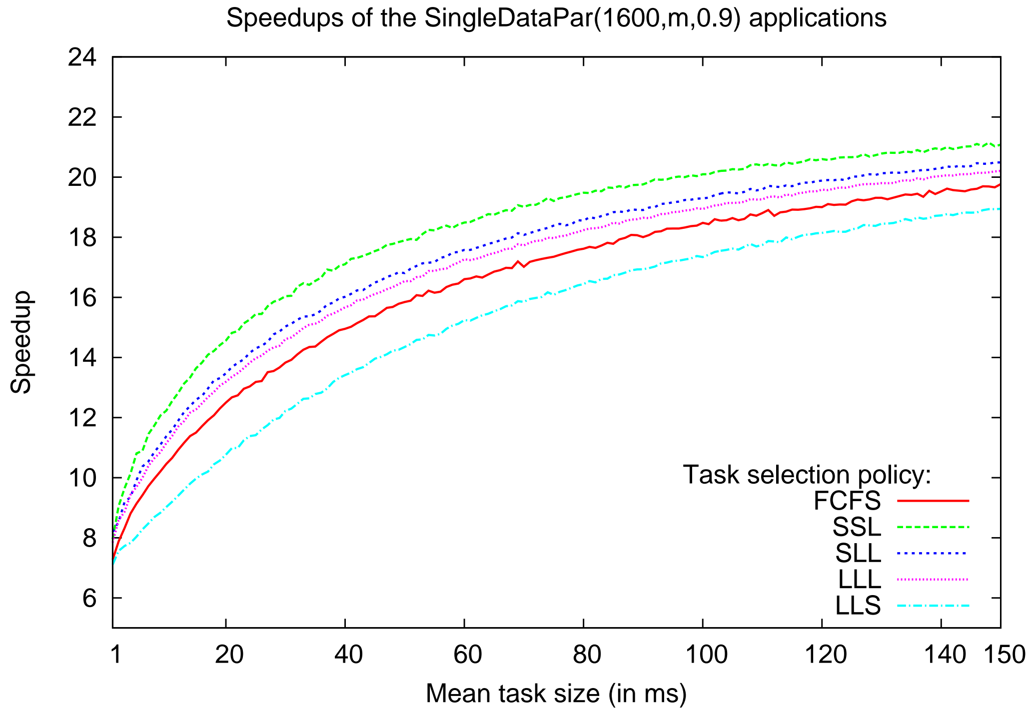
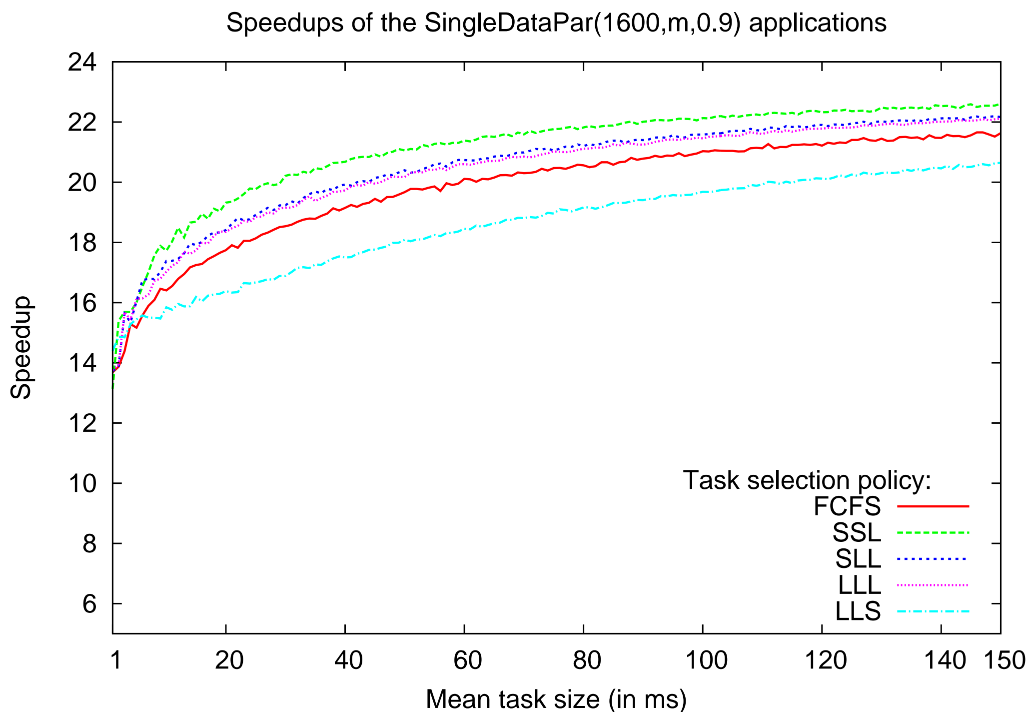


Figure 6.7: Speedups of the SingleDataPar(1600,m,0.9) applications on the Grid(2,8,0.5ms,30ms) environment

environment, where one cluster consists of 8 PEs and the other cluster consists of 16 PEs. Figure 6.8a shows the speedups when the main task is executed in a smaller (8 PEs) cluster, and the Figure 6.8b when it is executed on a larger (16 PEs) cluster. We can again observe the similar behaviour of the considered policies as on Figure 6.3, with the SSL policy giving the best speedups. However, we can see on Figure 6.8b that, in the case of the application's main task being executed on a larger cluster, the improvements that the SSL policy brings over FCFS are smaller. This is due to the fact that, under both policies, less tasks get to be offloaded to the PEs from the remote cluster (since the main cluster is larger, it will get much more work than the remote one) and, therefore, there is less chance for the SSL policy to bring improvements. We can also observe that there is very little difference between the SLL and LLL policies there – the difference is even smaller than on Figure 6.3. This is again due to the fact that the PEs from the main cluster execute more tasks than on the environment where both clusters consist of 8 PEs. This means that the main PE itself executes less tasks, so there is not much difference in whether it executes large or small tasks.



(a) The main task executed on the smaller cluster



(b) The main task executed on the bigger cluster

Figure 6.8: Speedups of the SingleDataPar(1600,m,0.9) applications on a two-clusters environment, with the latency between clusters being 30ms, and the latency inside clusters being 0.1ms. Cluster 1 consists of 8 PEs, whereas Cluster 2 consists of 16 PEs.

All of the applications that we have considered so far had the same number of tasks (100 per PE, which gives a total of 1600 tasks). However, no significant changes in the performance of the policies can be observed if we look at the applications with larger number of tasks. For example, Figure 6.9 shows the results equivalent to these on Figure 6.3 for the SingleDataPar(4800,m,0.9) and SingleDataPar(8000,m,0.9) applications. We can see that the relations between mean task size and performance of task selection policies is similar to the one on Figure 6.3, except that the speedups are generally higher (due to a larger number of parallel tasks).

Taking into account all of the experiments we have conducted in this section, we can make several conclusions about the performance of the granularity-driven task selection policies, when applied to highly irregular applications with a large number of tasks:

1. All of the granularity-driven task selection policies (except for LLS) bring improvements in the applications' speedups, when compared to the FCFS one. We can note the average speedup improvements of 10-20%, and up to 30% in the best case. It is always the case that the granularity-driven policies are better than the FCFS one.

2. The best policy to use is SSL.

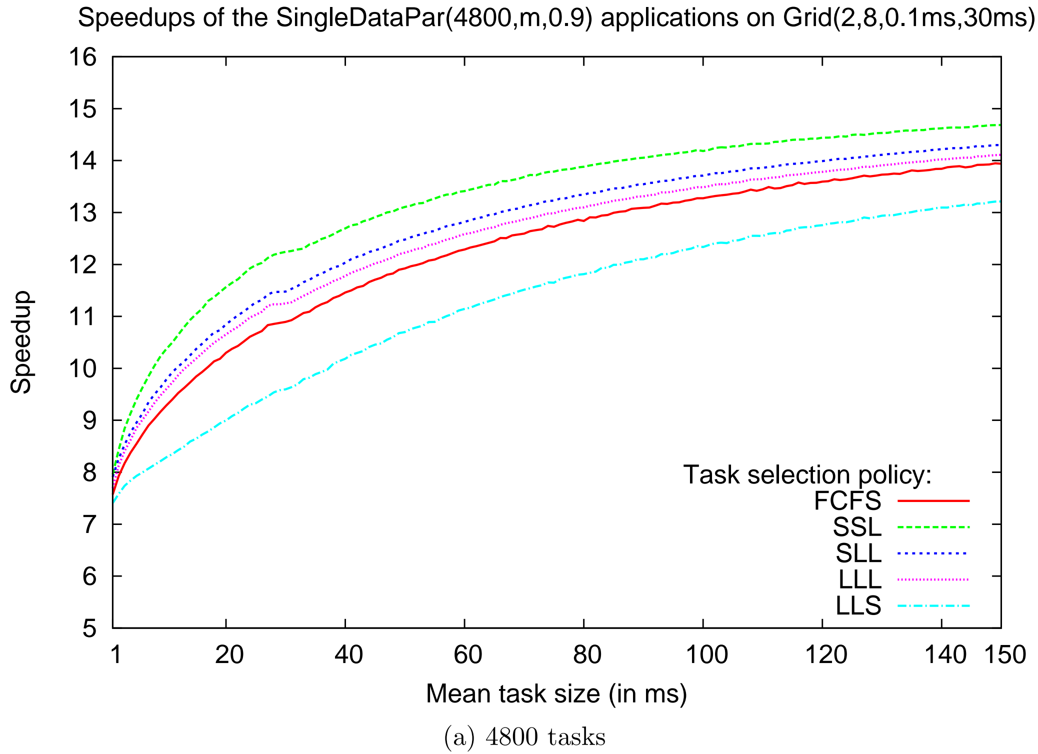
3. Generally, the strategy to leave the large tasks to the PEs in the remote cluster works the best. We can see significant improvements (up to 30%) with the policies that use this strategy (SSL, SLL, LLL) when compared to the LLS policy, which aims to execute the larger tasks locally and the smaller ones remotely.

4. Saving *the largest* tasks for the PEs from the remote cluster (under the SSL policy) brings small (but measurable) improvements compared to when only *large* (but not necessarily the largest) tasks are saved (in the case of the LLL and SLL policies).

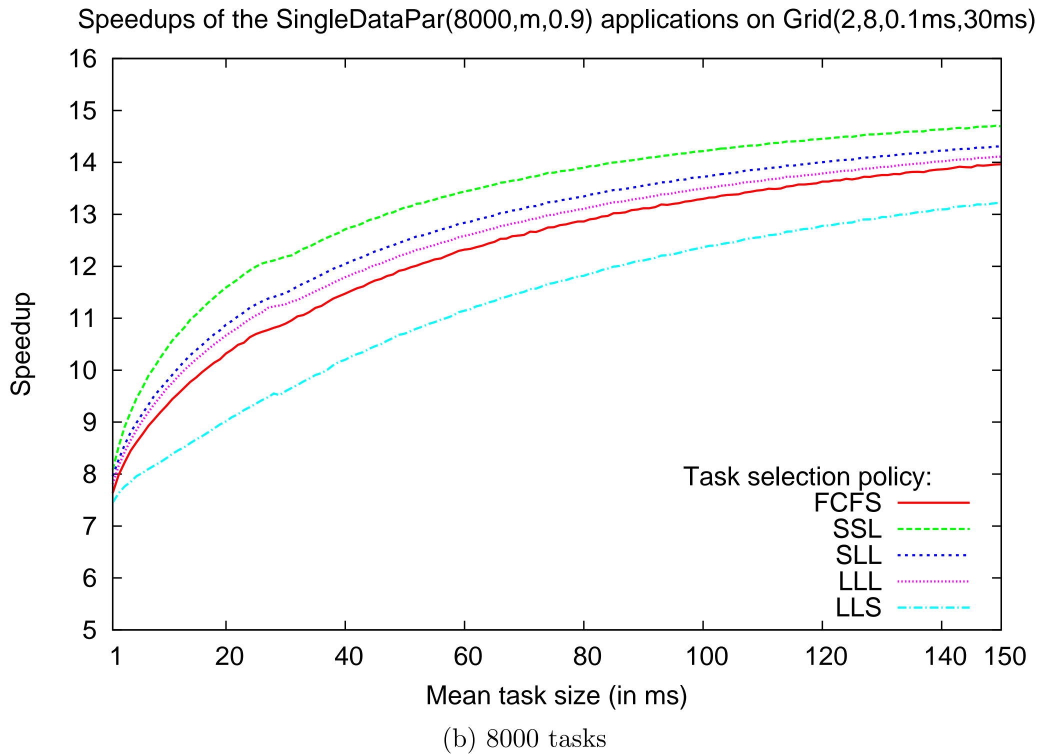
5. The improvements that the granularity-driven task selection policies bring are bigger for the applications with finer-grained tasks.

6.3.3 Applications with Variable Number of Tasks

In the previous section we have seen that the SSL task selection policy gives the best speedups for the applications with a larger number of tasks, no matter what the task granularity is. We have also seen that the improvements that this policy brings over



(a) 4800 tasks



(b) 8000 tasks

Figure 6.9: Speedups of the SingleDataPar(4800,m,0.9) and SingleDataPar(8000,m,0.9) applications on the Grid(2,8,0.1ms,30ms) environment

FCFS are in the range of 10-15% for most of the applications. In this section, we investigate how do the considered policies perform for the applications with a smaller number of tasks. We will again consider only highly-irregular applications (with the irregularity degree of 0.9). All of the applications we consider will be divided into three groups, based on the mean task granularity. Applications with very-fine grained tasks will be represented by the applications with the mean task size of 5ms, applications with fine-grained tasks by the applications with the mean task size of 30ms, and applications with coarse-grained tasks with the applications with the mean task size of 150ms. Again, as in the previous section, in most of the experiments we will consider our default computing environment, which is Grid(2,8,0.1ms,30ms).

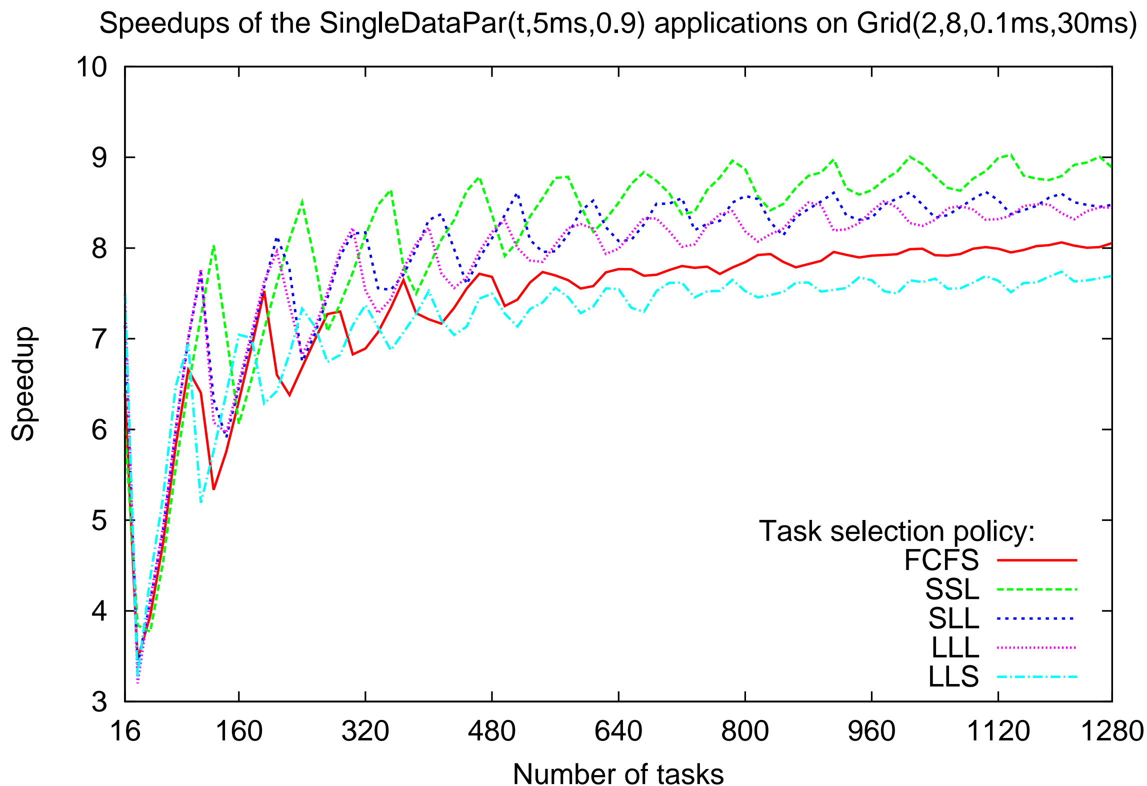


Figure 6.10: Speedups of the SingleDataPar(t,5ms,0.9) applications on the Grid(2,8,0.1ms,30ms) environment

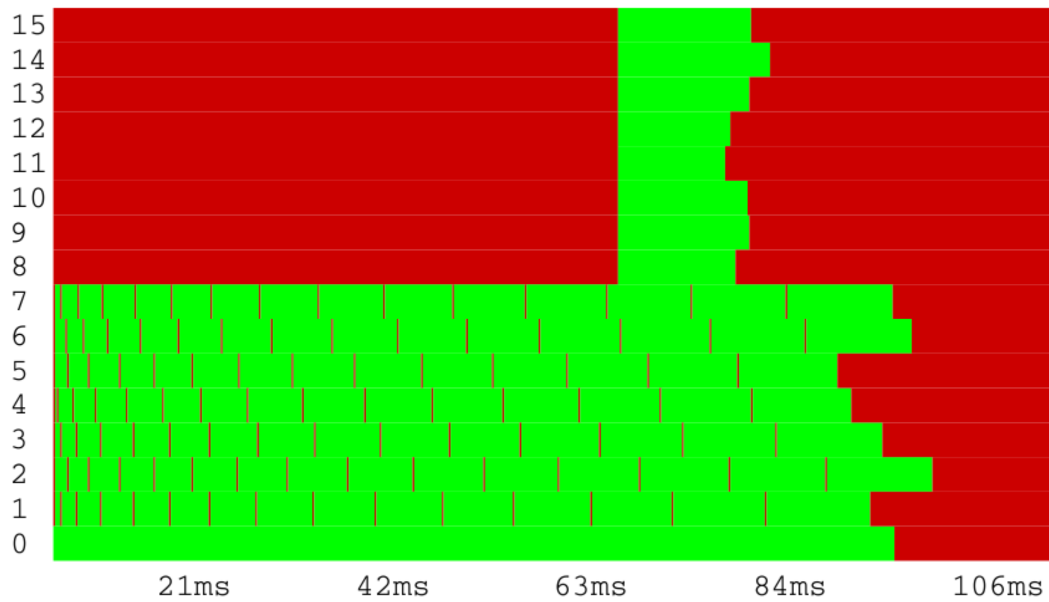
Figure 6.10 shows the speedups of the SingleDataPar(t,5ms,0.9) applications, where t is variable, under all of the considered task selection policies, executed on the Grid(2,8,0.1ms.30ms) environment. We can observe that the speedups are generally very variable. There is, however, certain regularity in the variations of speedups under individual task-selection policies. We can observe, for all policies, repeating phases

where the speedups first increase as the number of tasks in the applications increases, then reach the peak and after that, as the number of tasks in the application is further increased, start to decrease. This behaviour is especially notable for the applications with smaller number of tasks. The reason for this is as follows.

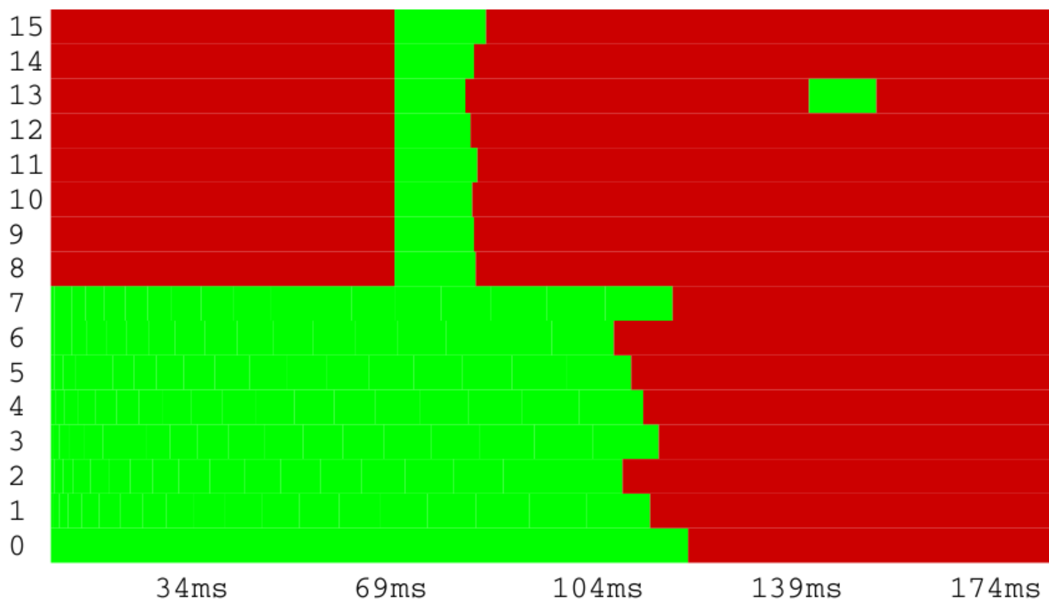
In the applications that we consider in this experiment, the task sizes are very small (relative to the communication latency between clusters), which means that the overall execution times of applications are also very small. Since the overheads of offloading tasks to the PEs from the remote cluster are very high (relative to the task sizes), speedups of these applications are very sensitive to the task placement decisions. When the application execution is approaching its end, a wrong decision about sending just one task from the main cluster to the remote cluster can create a perfect load imbalance, where the whole main cluster sits idle, waiting for a small task to be transferred to the remote cluster, executed there, and its result to be sent back to the main cluster. Taking into account that, on our default environment, the overhead in offloading a task to the remote cluster is 60ms, and that the mean task size is 5ms, it can happen that the decision to offload a single task to the remote cluster adds about 65ms to the application execution time, the same amount of time as if approximately 10 more tasks are executed on each PE in the main cluster! [2] This is the reason for the variations in the application speedups - under a fixed policy, for applications with a certain number of tasks, the described bad situation does not happen as PEs from the main cluster manage to grab last application tasks, and they do not have to wait for the remote PEs. However, when more tasks are added to the application, the remote PEs manage to grab some tasks near the end of the application execution, the described bad situation occurs and speedups start to decrease. Adding further tasks to the apllications again utilizes the PEs from the main cluster near the end of execution, and speedups again start to increase.

To illustrate this problem, consider Figure 6.11, which shows the activity profiles of example SingleDataPar(128,5ms,0.9) and SingleDataPar(160,5ms,0.9) applications under the SSL policy. We can see on Figure 6.10 that, for the SingleDataPar(128,5ms,0.9) applications, local maximum of speedup is obtained, while for SingleDataPar(160,5ms,0.9), local minimum is obtained. On Figure 6.11, the horizontal bars represent the utilization of individual PEs in the environment during the application execution, where the green parts are periods when the PE is busy executing tasks, and the red parts are periods where the PE is idle. For the SingleDataPar(128,5ms,0.9) application, we can observe that the PEs from the main cluster are not idle for long

[2] Note that this happens also for the applications with larger tasks, it is not reflected as badly on the applications' speedups as for the applications with smaller tasks.



(a) Activity profile under the SSL policy for SingleDataPar(128,5ms,0.9)



(b) Activity profile under the SSL policy for SingleDataPar(128,5ms,0.9)

Figure 6.11: Execution profiles of example SingleDataPar(128,5ms,0.9) and SIngleDataPar(160,5ms,0.9) applications under the SSL task-selection policy on the Grid(2,8,0.1ms,30ms) environment

time at the end of application execution. They have tasks to execute while the main PE is waiting for the remote PEs to send the results of the tasks they execute, so most of the PEs from the main cluster are idle just for the last 10ms or so of the application exeution. For the SingleDataPar(160,5ms,0.9) application, on the other hand, we can see that PE 13 manages to obtain a task near the end of the application execution, and that all of the PEs from the main cluster get idle very shortly after the transfer of this task is started. PEs from the main cluster then need to wait for this task to be transferred to PE 13, to be executed there, and for its result to be sent back to the main PE (PE 0). This results in most of the PEs from the main cluster being idle in the last 60ms of the application execution. This, of course, significantly decreases the overall application speedup, since the total application execution time is 174ms.

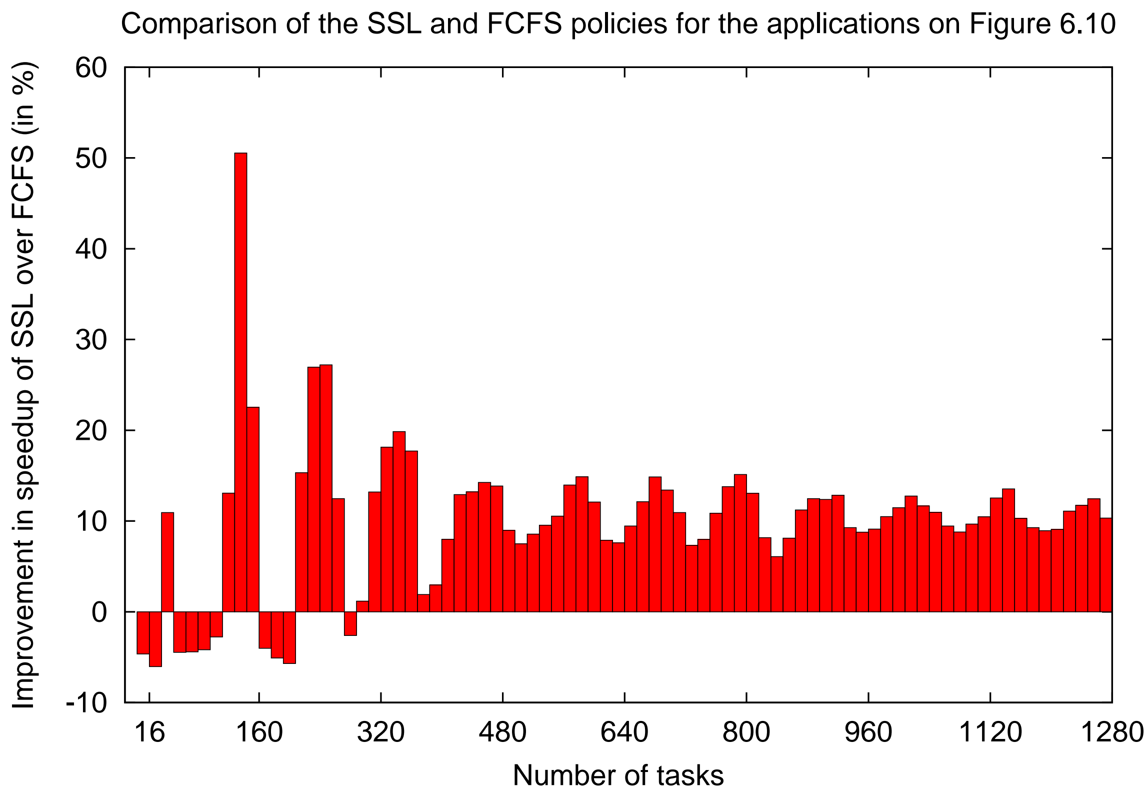


Figure 6.12: Improvements in speedups that the SSL policy brings over FCFS for the SingleDataPar(t,5ms,0.9) applications on the Grid(2,8,0.1ms,30ms) environment.

From Figure 6.10 we can observe that for the applications comprising a smaller number of tasks, all of the policies give similar speedup. As the number of tasks increases, the SSL and SLL policies start to outperform other policies. We can also observe that, in most of the cases, SSL performs better than SLL. For the applications

that comprise more than 700 tasks, SSL becomes a clear winner. Therefore, we can conclude that in most of the cases of applications with very-fine grained tasks, the SSL policy is the best one to use. However, in order to get the best performance for applications with any number of tasks, it would be necessary to have some kind of an adaptive policy, which would dynamically switch between the SSL and SLL policies, and in some situations altogether prevent offloading tasks outside of the cluster where they reside. We do not consider these improved policies in this thesis, as we leave this as a future work.

Figure 6.12 shows the speedup improvements that the SSL policy brings over FCFS for all applications considered on Figure 6.10. Again, we can observe the very variable results for the applications with smaller task sizes. In some cases, we can note the improvements of up to 50% in speedup, but in some other cases (for some applications with a very small number of tasks), the FCFS policy is actually better than SSL. As the applications get larger (i.e. when they comprise more tasks), the improvements that SSL policy brings become more regular, being between 10 and 20% for most of the applications.

Similarly to what we have done in the previous section, it would be useful to observe the speedup improvements that the best policy (SSL) brings over the worst one (LLS), to get the indication of how much improvements can we get in the best case, when the applications create tasks following a particularly "bad" pattern for FCFS policy. However, in the case of the applications with fine-grained tasks, we can see that speedups under the LLS policy are very similar to the ones under FCFS, so comparison between SSL and LLS would look very similar to the Figure 6.12. We will, therefore, omit this comparison for this class of applications.

Figure 6.13 shows the speedups of the applications with the mean task size of 30ms. For the applications comprising a small number of tasks, we can see a similar situation as on Figure 6.10. The speedups are variable, but the SSL policy gives the best speedups for most of the applications. When the number of tasks in the application is larger than 160, the SSL policy clearly outperforms all others. We can also see that, for the applications comprising a larger number of tasks, we get much less variations in speedups, since placing a few tasks on wrong PEs does not have as bad impact on application's speedup as in the case of applications with very-fine grained tasks.

Figure 6.14 shows the speedup improvements that the SSL policy brings over both the FCFS and the LLS policy for the applications considered on Figure 6.13. We can observe that the improvements over the FCFS policy (Figure 6.14a) are in the range of 10-15% for the applications comprising a larger number of tasks. Again, as on Figure

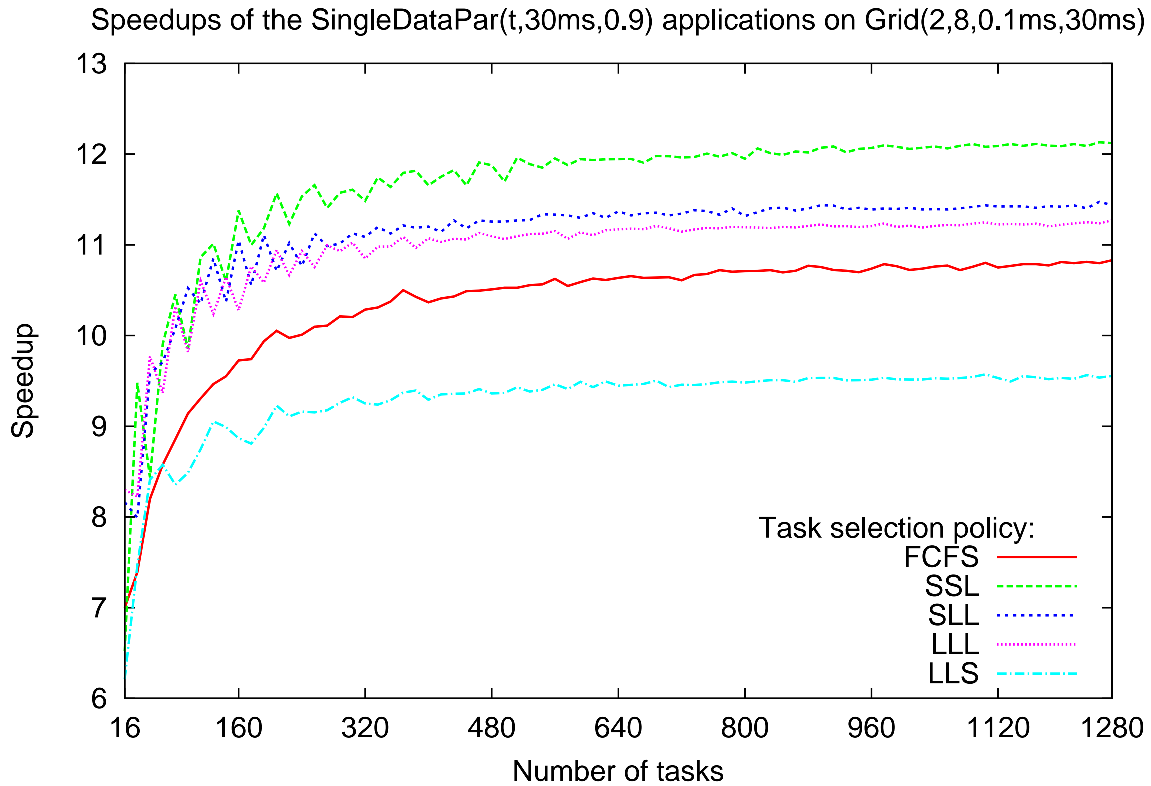
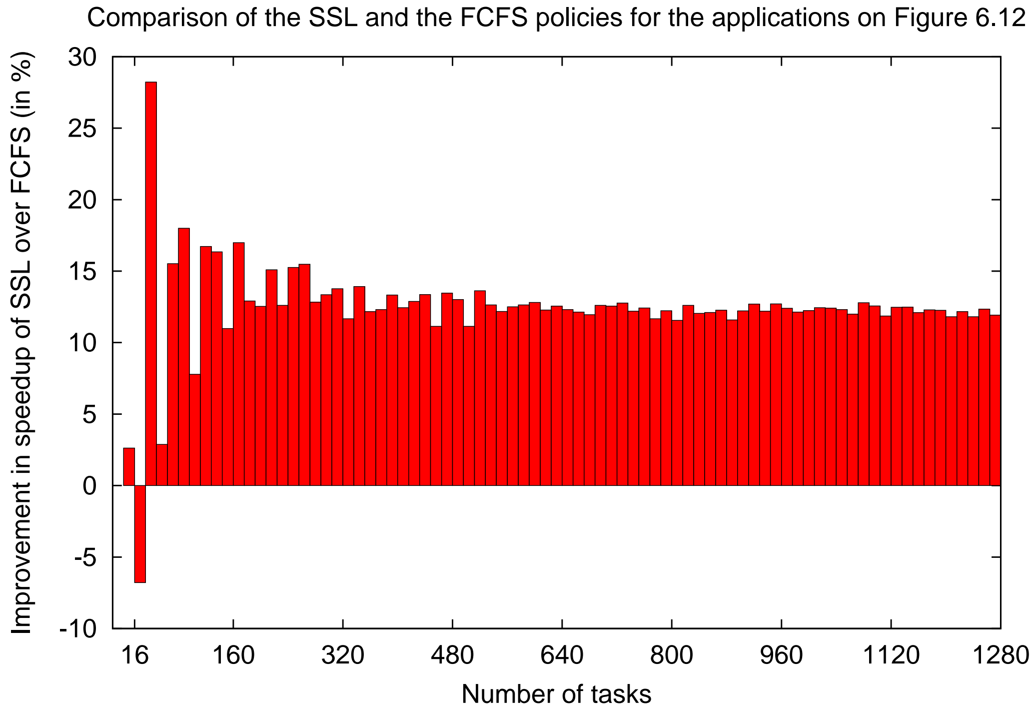


Figure 6.13: Speedups of the SingleDataPar(t,30ms,0.9) applications on the Grid(2,8,0.1ms,30ms) environment

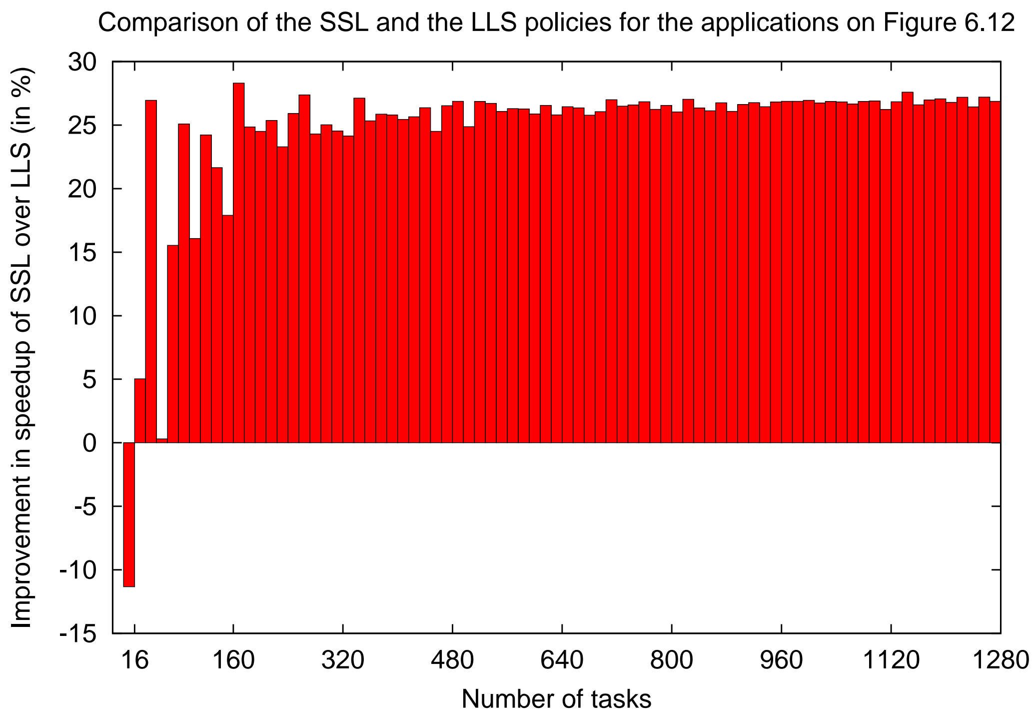
6.12, the speedup improvements for applications comprising a smaller number of tasks are highly variable, ranging from -6% to 28%. When we compare the SSL with the LLS policy (Figure 6.14b), we can observe that the improvements are between 25 and 30% for almost all of the applications (again, the exceptions are applications comprising a very small number of tasks).

Figure 6.15 shows the speedups of the applications with the mean task size of 150ms. Here, we can see that for applications comprising a smaller number of tasks, the LLL and SLL policies notably outperform all others, and for the applications comprising a larger number of tasks, the SSL policy gives the best speedups. This indicates that, for highly irregular applications with coarse-grained tasks, that comprise a small number of tasks, the greedy strategy to try to execute the largest task as soon as possible is the best one.

Figure **??**, similarly to Figure 6.14, shows the improvements that the SSL policy brings over the FCFS and LLS ones. We can observe that the improvements in both cases are small (around 5% over FCFS, and around 5-10% over LLS, except for the ap-



(a) Improvements of SSL over FCFS



(b) Improvements of SSL over LLS

Figure 6.14: Improvements in speedups under the SSL policy over FCFS and LLS for the SingleDataPar(t,30ms,0.9) applications on the Grid(2,8,0.1ms,30ms) environment

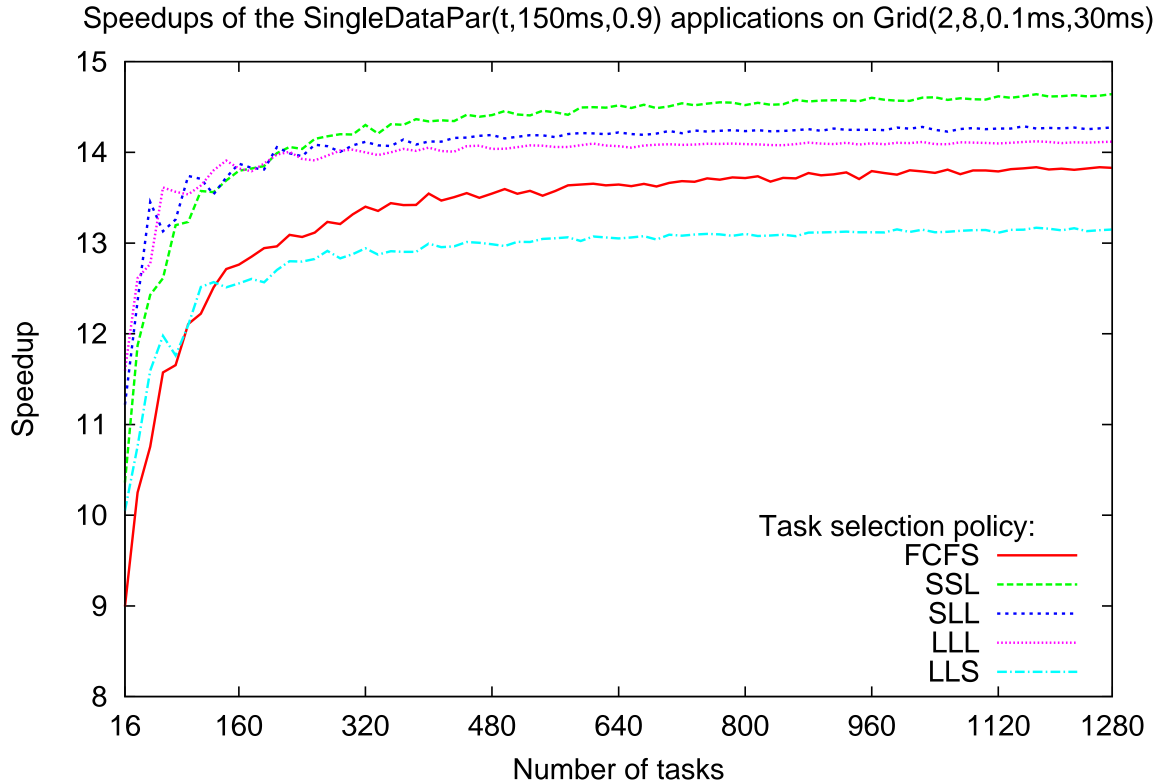
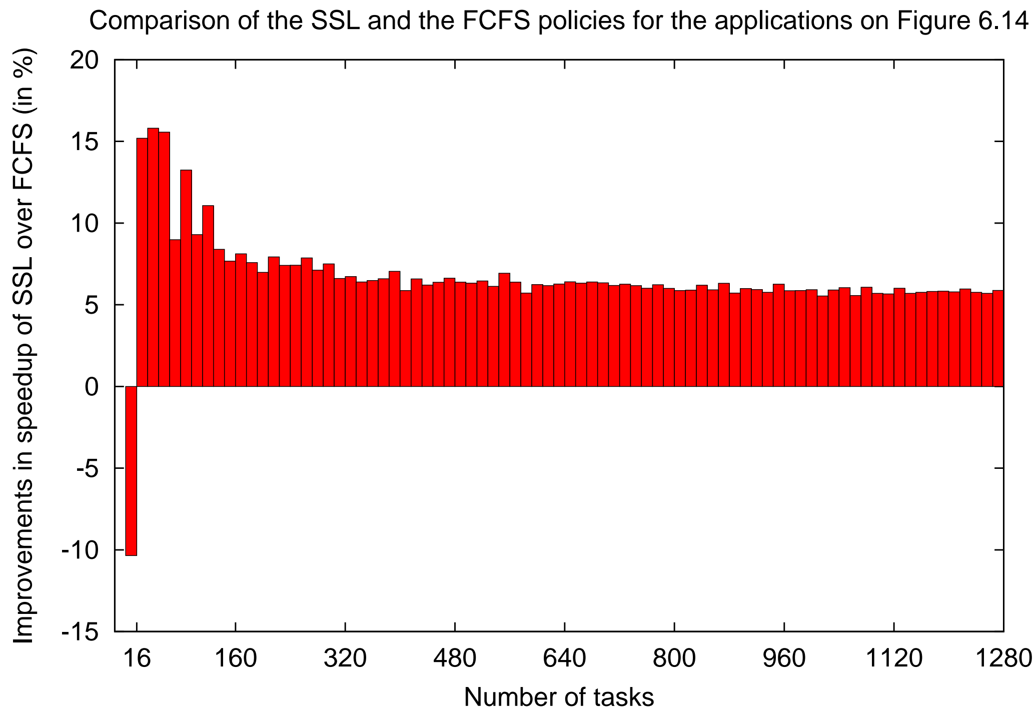


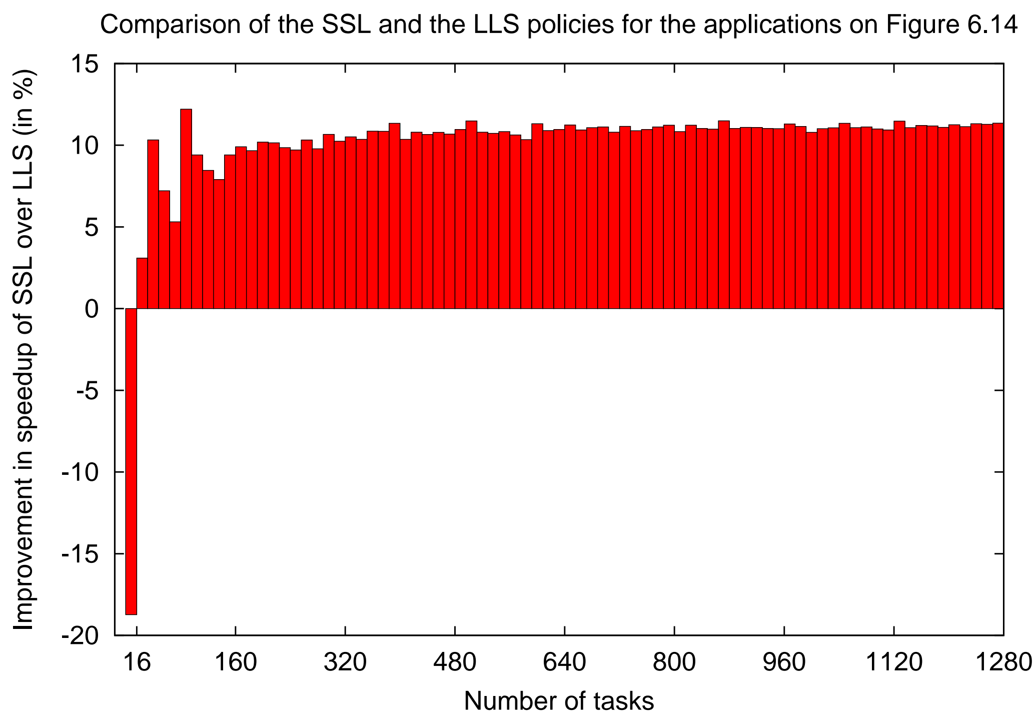
Figure 6.15: Speedups of the SingleDataPar(t,150ms,0.9) applications on the Grid(2,8,0.1ms.30ms) environment

plications comprising a very small number of tasks), but still measurable. In addition, since we have observed that, for the applications comprising a smaller number of tasks, the LLL and SLL policies are the best (and the speedups under these two policies are very similar most of the time), Figure 6.17 shows the speedup improvements that the LLL policy brings over FCFS for these applications. We can observe that in this case the improvements are notably better (for the applications comprising up to 100 tasks, improvements are between 15 and 30%) than in the case of the SSL policy.

From all of the experiments in this section, we can conclude that the SSL policy gives the best speedups for all the applications we have considered, except for the ones that comprise a small number of tasks, and where tasks are very-fine or very-coarse grained. In the case of the applications with very fine-grained tasks, the application execution times under all of the policies are very small, so any policy is probably a good choice. For the applications with coarse-grained tasks, it is worthwhile to use the LLL or the SLL policy instead of the SSL one, provided that the number of tasks in an application is small.



(a) Improvements of SSL over FCFS



(b) Improvements of SSL over LLS

Figure 6.16: Improvements in speedups under the SSL policy over FCFS and LLS for the SingleDataPar(t,150ms,0.9) applications on the Grid(2,8,0.1ms,30ms) environment

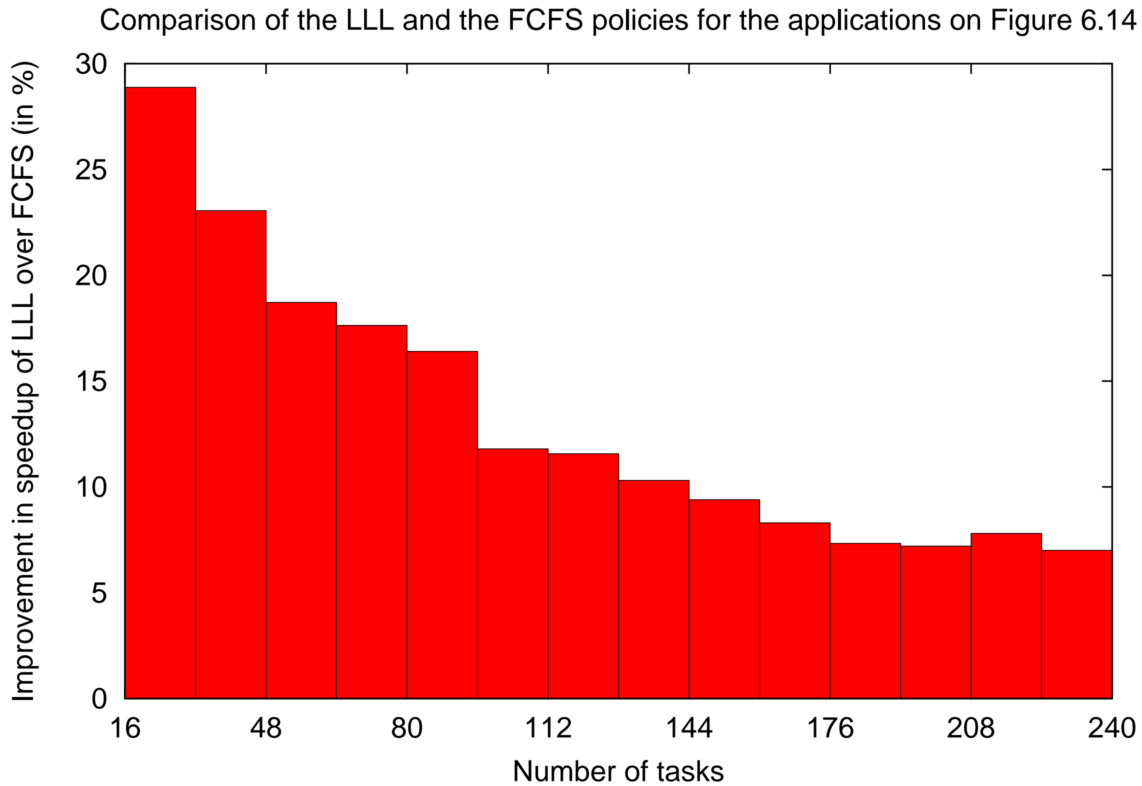
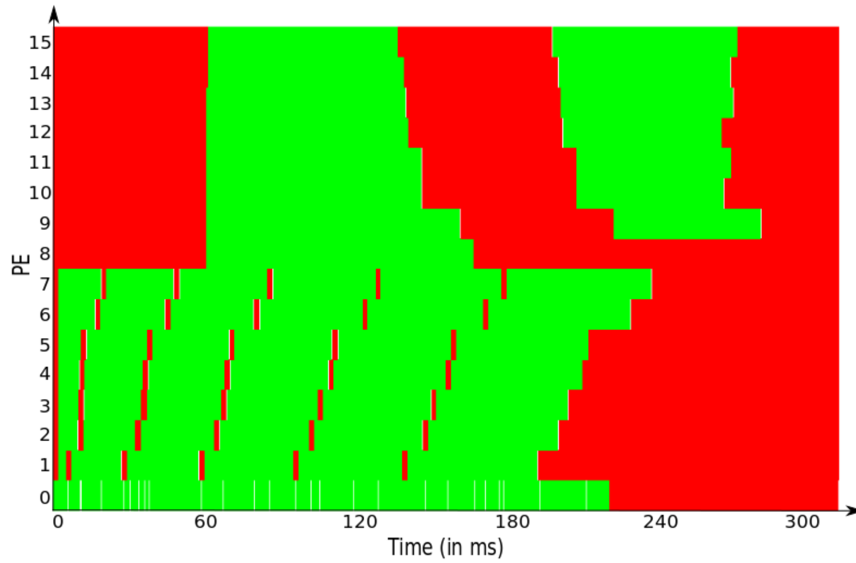


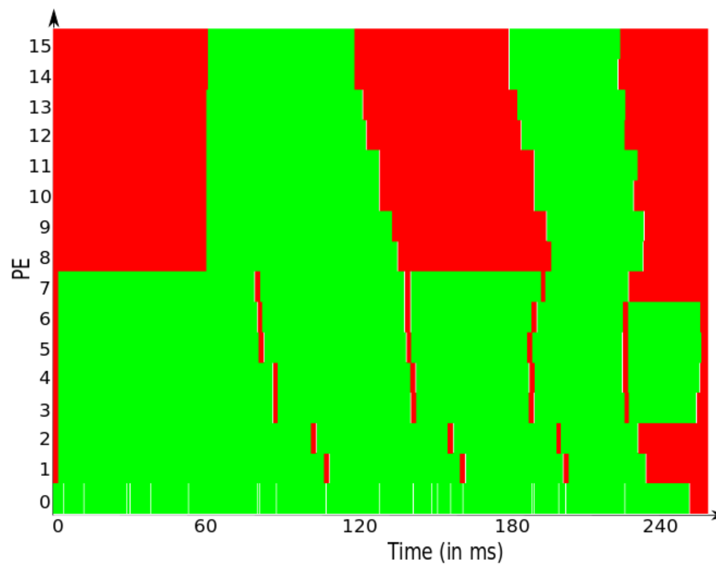
Figure 6.17: Improvements in speedups under the LLL policy over FCFS for the SingleDataPar(t,150ms,0.9) applications on the Grid(2,8,0.1ms.30ms) environment

To see why, for the applications comprising a smaller number of tasks, the SLL policy outperforms the SSL one, we can note the following. For these applications, using the SLL policy means that the main PE grabs the smallest tasks, the PEs from the main cluster grab the largest ones (since they will be the first ones to obtain work from the main PE), leaving the medium ones to the PEs from the remote cluster. In this way, the application execution does not get stalled when the PEs from the main cluster do not have any work to do, and the main PE is waiting to obtain the results of large tasks that are being executed on the remote cluster. This problem occurs under the SSL policy, as under that policy the PEs from the main cluster will quickly finish executing the smallest tasks, and these PEs will not have any work to do while the PEs from the remote cluster are executing larger tasks (similar situation that we have already described when discussing the performance issues of the policies for the applications with very-fine grained tasks).

This can be seen on Figure 6.18, which shows the activity profiles of an example SingleDataPar(48,150ms,0.9) application. We can see that in the case of the SSL



(a) SSL activity profile



(b) SLL activity profile

Figure 6.18: Profiles of run of an example SingleDataPar(48,150ms,0.9) application under the SSL and SLL task selection policies

policy, the PEs from main cluster (PEs 0-7) finish executing small tasks (at time of about 240ms) and are then idle at the end of the application execution, while the PEs from the remote cluster (PEs 8-15) are executing larger tasks and sending their results back. We can also see that this does not happen in the case of the SLL policy. The PEs from the main cluster will grab fewer large tasks at the beginning of the application execution, which means that, at the end of the execution, the main PE will still have tasks to offload to both the main and the remote cluster. Furthermore, the tasks that get sent to the remote cluster at the end of the application execution are not large, so less time is spent in waiting for their completion and sending their results back. This results in speedup which is about 20% better than in the case of the SSL policy. This is, perhaps counter-intuitive, the situation where sending the largest tasks to the PEs from the remote cluster is not necessarily the best thing to do.

For the applications comprising a larger number of tasks, the benefits of keeping the PEs from the remote cluster busy for more time (so that they need to look for work over high-latency network less often), by sending them the largest tasks, far outweighs the potential drawbacks of the PEs from main cluster being idle at the end of application execution. Therefore, for this kind of applications, the SSL policy performs better than both the SLL and the LLL one.

To summarise our findings from the experiments presented in this section, we can conclude the following:

1. For all of the applications that we have considered, except for those comprising a small number of tasks, the SSL policy gives the best speedups for all mean task sizes. The speedup improvements that this policy brings over the FCFS one are similar to those noted for the applications comprising a larger number of tasks in the Section 6.3.2 – between 10% and 20% on the average, and 20-30% in the best case.

2. For the applications comprising a smaller number of tasks, there is no clear winning policy. In the case of the applications with coarse-grained tasks, the best policies are SLL and LLL. For the applications with fine-grained tasks, all of the granularity-driven task selection policies (except for the LLS one, which is constantly the worst one) give the similar speedup.

3. Except in the case of the applications that comprise very small number of very-fine grained tasks, it is always the case that the best granularity-driven task selection policy gives the better speedup than the FCFS policy. Therefore, except in the mentioned case, having information about the task sizes is always

beneficial.

4. In order to develop a single granularity-driven task selection policy that will give the best speedups for all applications, it is necessary to develop an adaptive policy, which will dynamically switch between the SSL, SLL and LLL policies. It would use SSL policy whenever the number of remaining parallel tasks in the application is above some application-specific threshold, and when it falls below, it would switch to either SLL and LLL, or even to FCFS in the case when the number of remaining tasks is very small, and they are all very-fine grained.

6.3.4 Applications with a Varying Degree of Irregularity

One of the purposes for introducing the definition of the application's degree of irregularity in Section 4.2 was to test the hypothesis that the granularity-driven task selection policies bring better speedup improvements (compared to the policies that do not use information about the task sizes) for more irregular applications. So far, we have only studied very irregular applications, with the degree of irregularity being 0.9. In this section, we investigate how the speedup improvements that granularity-driven task selection policies bring change as the degree of irregularity of an application increases from 0.0 to 0.9.

Similarly to Section 6.3.3, we will consider three main classes of applications : the applications with mean task size of 5ms (the applications with very fine-grained tasks), 30ms (the applications with fine-grained tasks) and 150ms (the applications with coarse-grained tasks). For each type of application (with the fixed mean task size, the number of tasks and the degree of irregularity), we tested 100 different applications with randomly generated task sizes to obtain minimal, average and maximal speedup improvements (in percentages) that the granularity-driven task selection policies bring over FCFS. For all of the experiments in this section, we assume that the applications are executed on our default computing environment (Grid(2,8,0.1ms,30ms)).

Figure 6.19 shows the speedup improvements that the SSL policy brings over FCFS for the applications with very-fine grained tasks (SingeDataPar(1600,5ms,d), where d varies from 0.1 to 0.9). The three bars in the figure show the minimal (red), the average (green) and the maximal (blue) speedup improvement obtained over 100 applications of the same type (i.e. with the same mean task size, the degree of irregularity and the number of tasks). We can observe that the SLL policy brings improvements to the speedups for all considered applications. We can also clearly observe that, as the application irregularity increases, the minimal, average and maximal speedup improve-

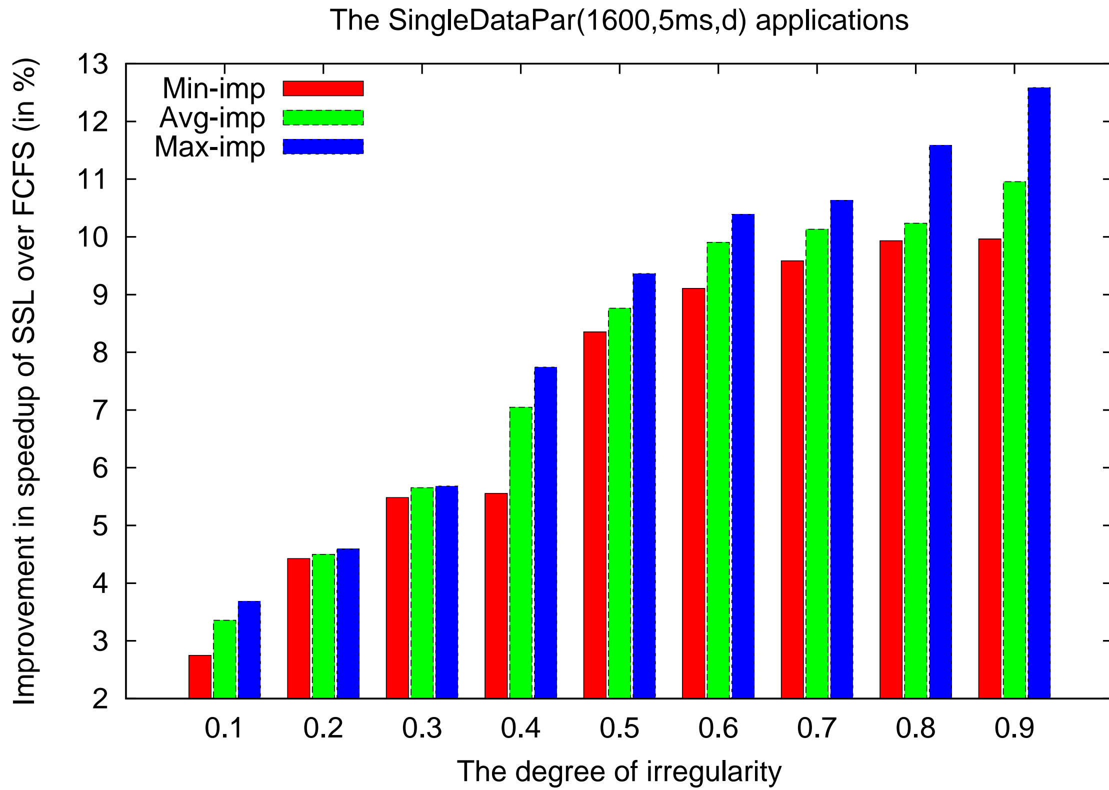


Figure 6.19: Improvements that the SLL policy brings over FCFS for the SingleDataPar(1600,5ms,d) applications on the Grid(2,8,0.1ms,30ms) environment

ments that the SLL policy brings all increase from 2-4% (for the applications with the degree of irregularity of 0.1) up to 10-13% (for the applications with the degree of irregularity of 0.9). We can also see that in most of the cases, the improvements are pretty uniform – there is not much difference between the minimal, average and the maximal improvement. This indicates that there are no especially good or especially bad cases of applications of this particular kind for the SSL policy – the improvements it brings over FCFS are more or less constant for all applications.

Figure 6.20 shows the improvements in speedups that the SSL policy brings over FCFS for the SingleDataPar(1600,30ms,d) applications, where d again varies from 0.1 to 0.9. Again, we can see that the minimal, average and maximal improvements are constantly increasing as the application's irregularity increases, and that there is even less difference between the minimal, average and maximal improvement than on Figure 6.19. The same conclusion we get when we observe the applications with coarse-grained tasks, shown on Figure 6.21, where the SingleDataPar(1600,150ms,d) applications are considered. Although the improvements here are more modest than

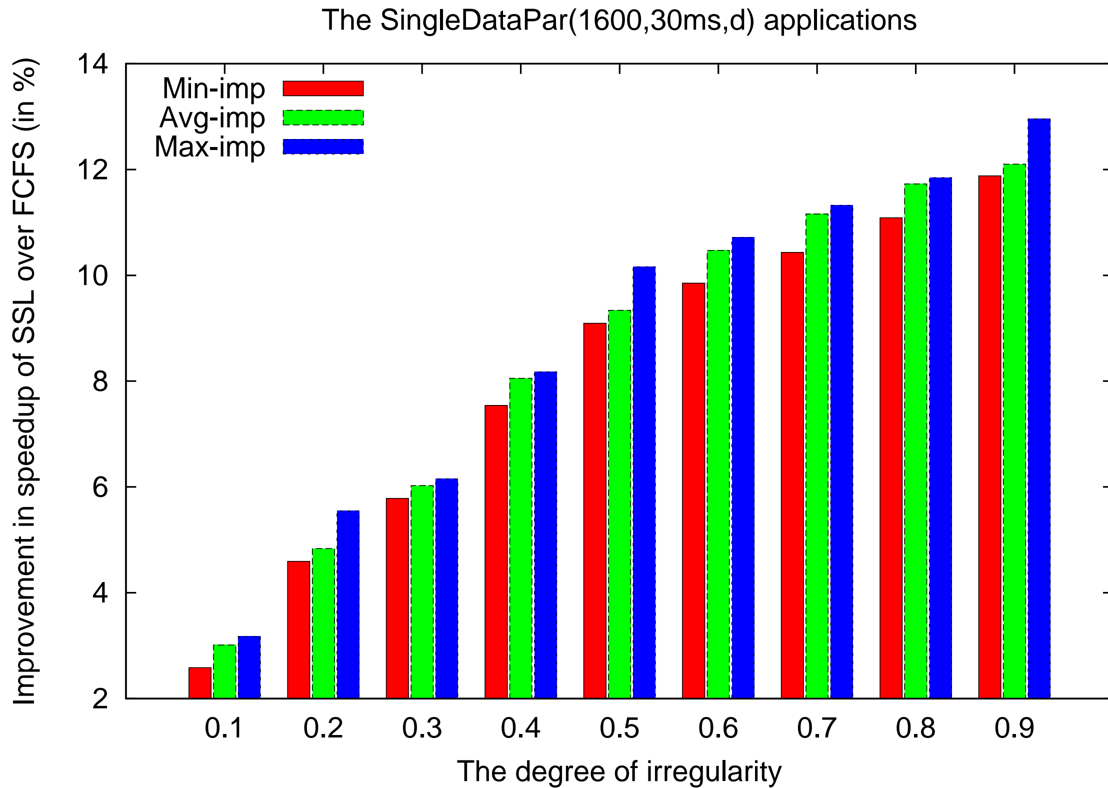


Figure 6.20: Improvements that SLL brings over FCFS for SingleDataPar(1600,30ms,d) applications on Grid(2,8,0.1ms,30ms) environment

in the previous two cases, we can still see that they are increasing as the application's degree of irregularity increase.

Finally, Figure 6.22 shows the improvements for the applications that comprise a smaller number of tasks (48 in this case), with the mean task size of 150ms. In the Section 6.3.3, we have observed that, for this kind of applications, the SLL and LLL policies give the best speedups. On Figure 6.22 we, therefore, compare the LLL and FCFS policies. As on the previous figures, here we can also observe that the improvements are increasing as the application's degree of irregularity increases. We can also observe a bigger difference between the average and the maximal speedup improvements that the best policy (in this case, LLL) bring over FCFS than in the case of Figures 6.19, 6.20 and 6.21, especially for more irregular parallel applications. Again, this is due to the applications comprising a smaller number of tasks being more sensitive to task placement decisions, and therefore FCFS can perform especially badly for some of them. LLL policy is less prone to offloading the 'wrong' tasks to the PEs from the remote cluster for this kind of applications, because typically these PEs will

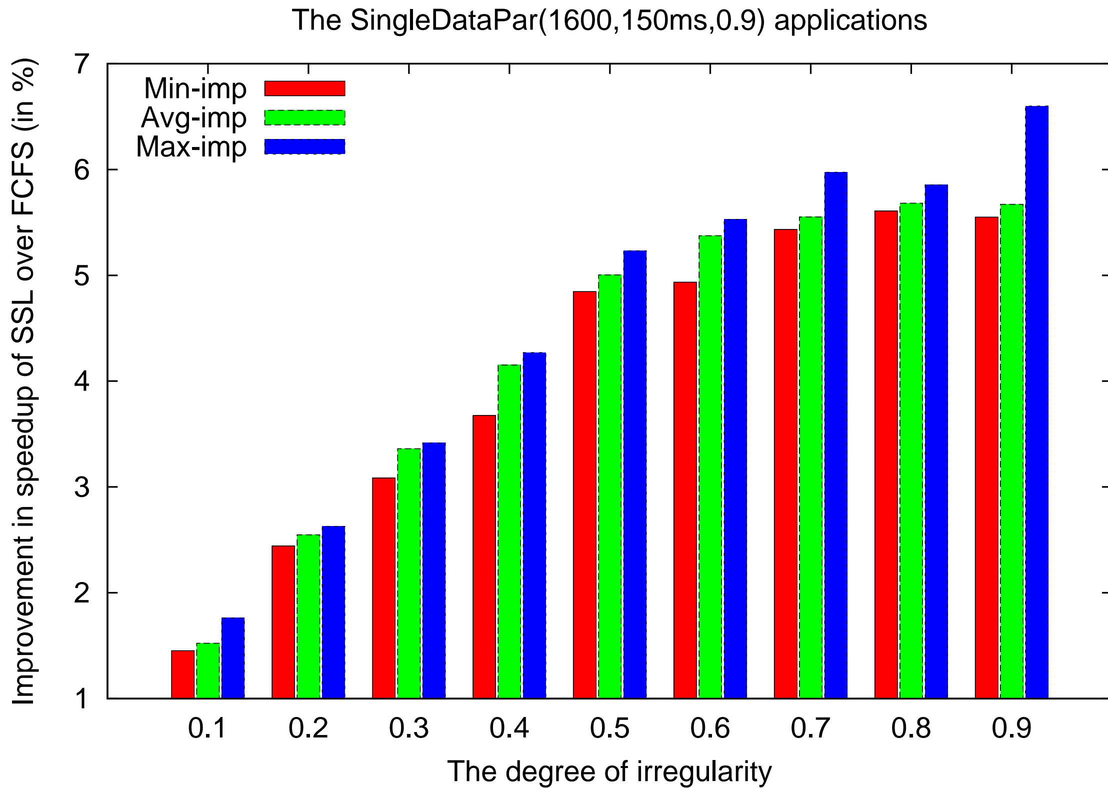


Figure 6.21: Improvements that the SLL policy brings over FCFS for the SingleDataPar(1600,150ms,d) applications on the Grid(2,8,0.1ms,30ms) environment

receive smaller tasks (as opposed to, possibly large, random tasks under the FCFS policy), which is the right decision for this type of applications.

From the experiments in this section, we can clearly conclude that the granularity-driven task selection policies bring better speedup improvements (compared to the FCFS one) for more irregular applications.

For the applications comprising a larger number of tasks, we can also conclude that there is not much variability in improvement that these policies bring when we consider different applications which have the same mean task size, the number of tasks and the degree of irregularity (i.e. the minimal, average and maximal improvements when different applications are considered are more or less the same). In other words, the amount of improvement is not dependent on the actual task sizes, but rather just on the *mean* task size and the degree of irregularity (together with the number of tasks, of course). This further means that we do not need to know all of the details about all of the tasks of the application being executed in order to be able to estimate how much improvement can we get with the granularity-driven task selection policies. We,

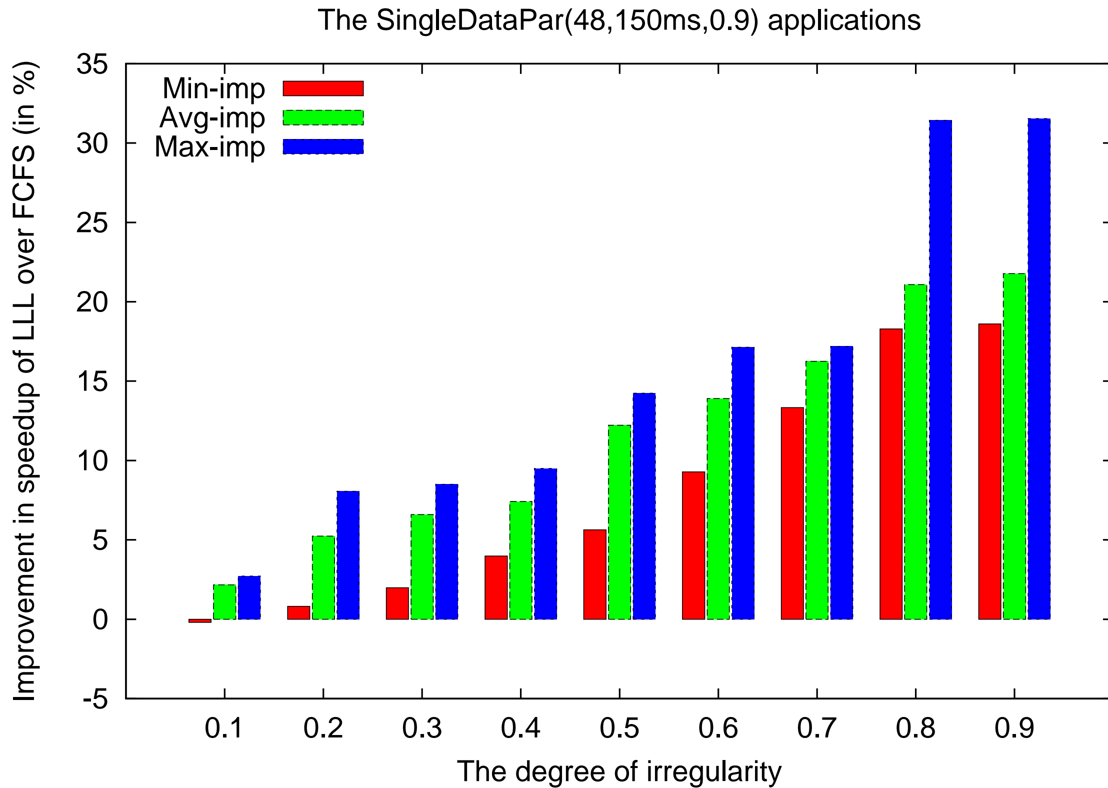


Figure 6.22: Improvements that the LLL policy brings over FCFS for the SingleDataPar(48,150ms,d) applications on the Grid(2,8,0.1ms,30ms) environment

rather, need to know just the application's overall parallel profile, in terms of how different its tasks are and what the mean task size of the application is.

For the applications comprising a smaller number of tasks, there are bigger differences between the minimal, average and maximal improvement that the best policy (LLL) brings over FCFS (although, as we have noted before, all of them are better the more irregular the application is). This means that for this type of applications, we cannot precisely estimate what improvements we will get from the mean task size and the degree of irregularity alone, although we still know that the improvements are going to be significant, especially for more irregular applications.

6.3.5 Computing Environments with a Hierarchy of Latencies

So far, we have only considered the computing environments that comprise two clusters, connected by a high-latency network. We will now investigate how do the granularity-driven task selection policies perform for the environments comprising multiple clusters, where the latencies between all clusters are not uniform. We will consider

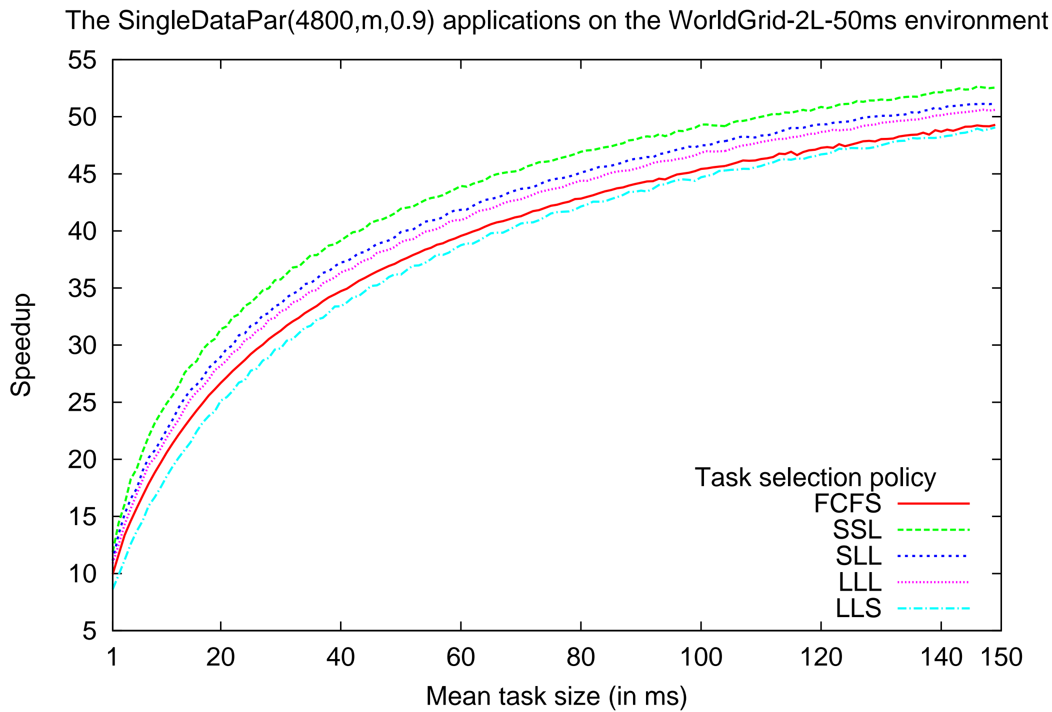
Comp. environment	**Description**
WorldGrid-Hom	Homogeneous system, latency between every two PEs is 0.1ms
WorldGrid-Uni-10ms	Latency between every two clusters is 10ms (Same as Grid(8,8))
WorldGrid-2L-20ms	Clusters split into two continental groups of 4 clusters each, latency between them being 20ms. Within each continental group, latency between every two clusters is 10ms
WorldGrid-2L-30ms	Same as WorldGrid-2Levels-20ms, but with latency between continental groups being 30ms
WorldGrid-2L-50ms	Same as WorldGrid-2Levels-20ms, but with latency between continental groups being 30ms
WorldGrid-3L-80ms-30ms	Clusters split into two continental groups of 4 cluster each, latency between them being 80ms. Each continental group is split into two country groups (with 2 clusters in each group), with the latency between them being 50ms. Each country group split into two site-clusters, latency between them being 10ms (See Figure 5.2)

Figure 6.23: The WorldGrid computing environments simulated. Each environment consists of 8 clusters, each of which consists of 8 PEs

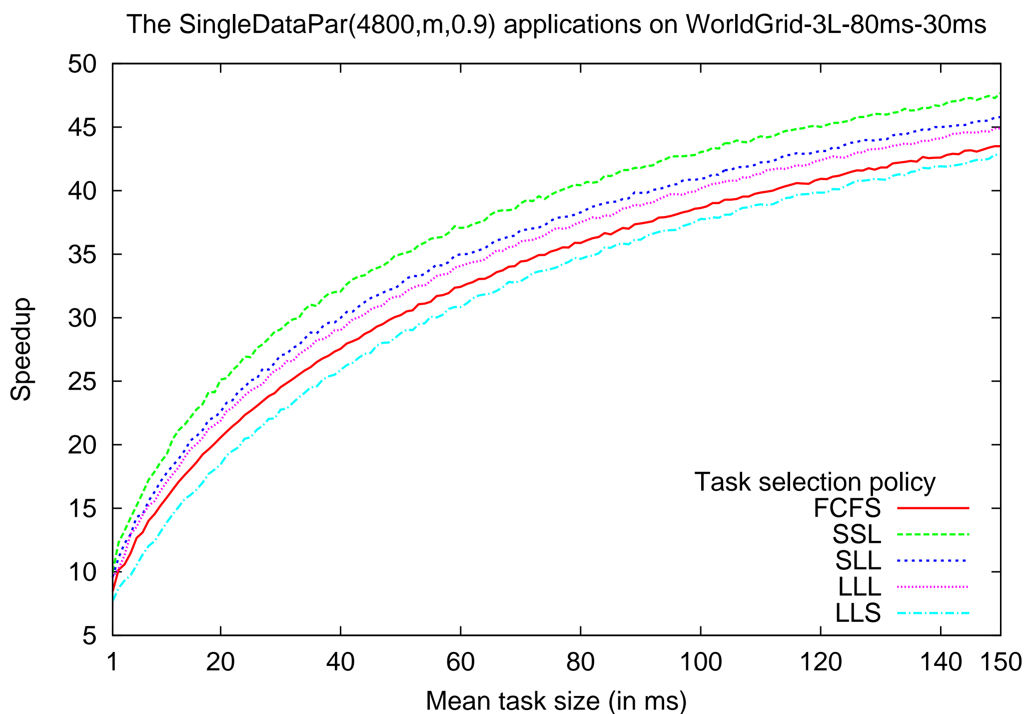
the same example environments that we used when evaluating the usability of perfect load information in work-stealing, in Section 5.2. As a reminder, Table 6.23 lists the considered environments.

In our first experiment, we focus on the applications with a fixed degree of irregularity, a fixed number of tasks and variable mean task size (i.e applications considered in Section 6.3.2). Figure 6.24 shows the speedups for the SingleDataPar(4800,m,0.9) applications on the WorldGrid-2L-50ms and WorldGrid-3L-80ms-30ms computing environments. We can observe a similar behaviour of all of the considered policies on both figures – the SSL policy constantly gives the best speedups, and the LLS one gives the worst. These results are very similar to the ones that we obtained for the same class of applications on various two cluster environments (Section 6.3.2).

Very similar results to these shown on Figure 6.24, in terms of the relations between speedups under different policies, can also be obtained for other WorldGrid computing environments. Nothing particularly changes as the latency between clusters change – the SSL policy gives the best speedups, followed by the SLL and LLL policies. The FCFS policy gives worse speedups that the SSL, SLL and LLL policies, but also is



(a) WorldGrid-2L-50ms



(b) WorldGrid-3L-80ms-30ms

Figure 6.24: Speedups of SingleDataPar(4800,m,0.9) applications on WorldGrid-2L-50ms and WorldGrid-3L-80ms-30ms computing environments

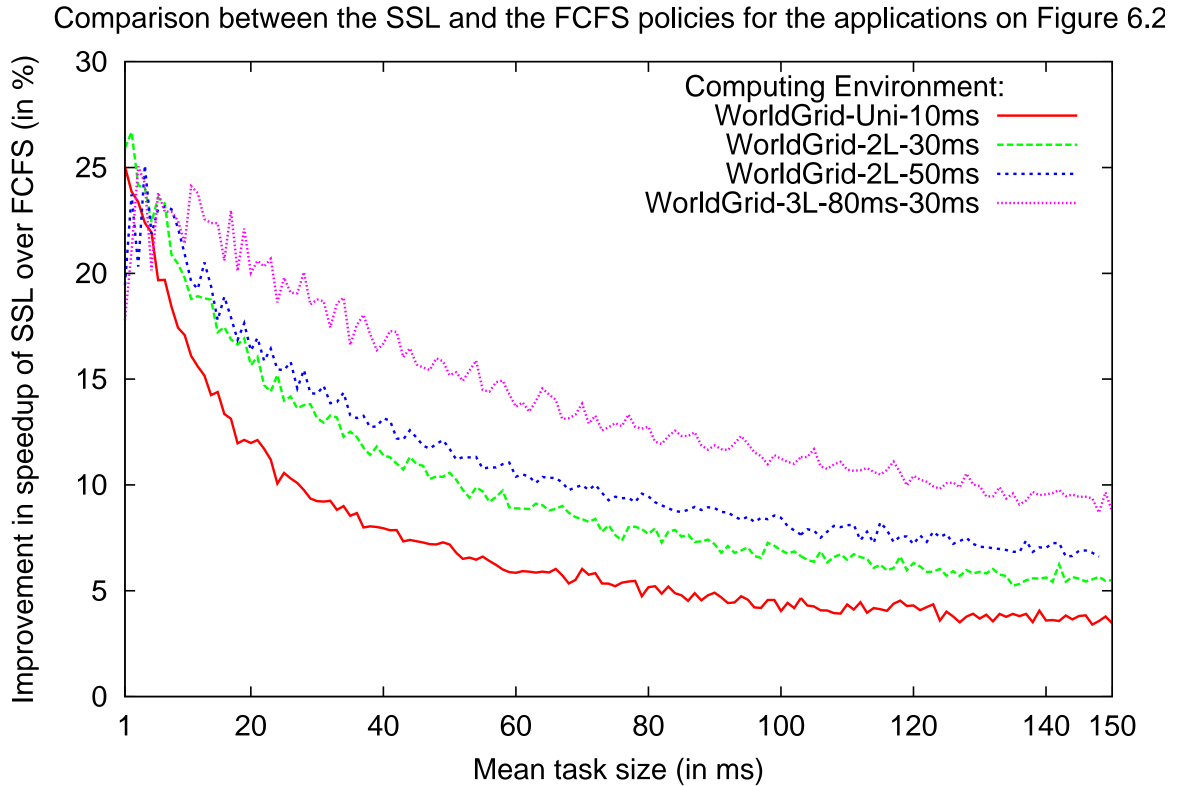


Figure 6.25: Improvements that SLL brings over FCFS for SingleDataPar(4800,m,0.9) applications on various WorldGrid environments

constantly better than LLS. We do not show these results here. Instead, we will focus on the amount of speedup improvement that the SSL policy brings on various WorldGrid environments, to observe how does this change as the environment gets more heterogeneous.

Figure 6.25 compares the speedup improvements under the SSL policy over the FCFS one for the applications considered on Figure 6.24, on various WorldGrid computing environments. We can clearly observe that the improvements are increasing as we consider more heterogeneous environments with higher communication latencies (except for the applications with very-fine grained tasks, in which case the better improvements are on more homogeneous environments). For the kind of applications we are considering here (i.e. single-level data parallel applications), however, the main reason for better improvements on more heterogeneous environments is the presence of higher communication latencies in them. Since tasks only get transferred between the main cluster and the other clusters, the fact that the latencies between different clusters are different does not have significant impact on the performance of task se-

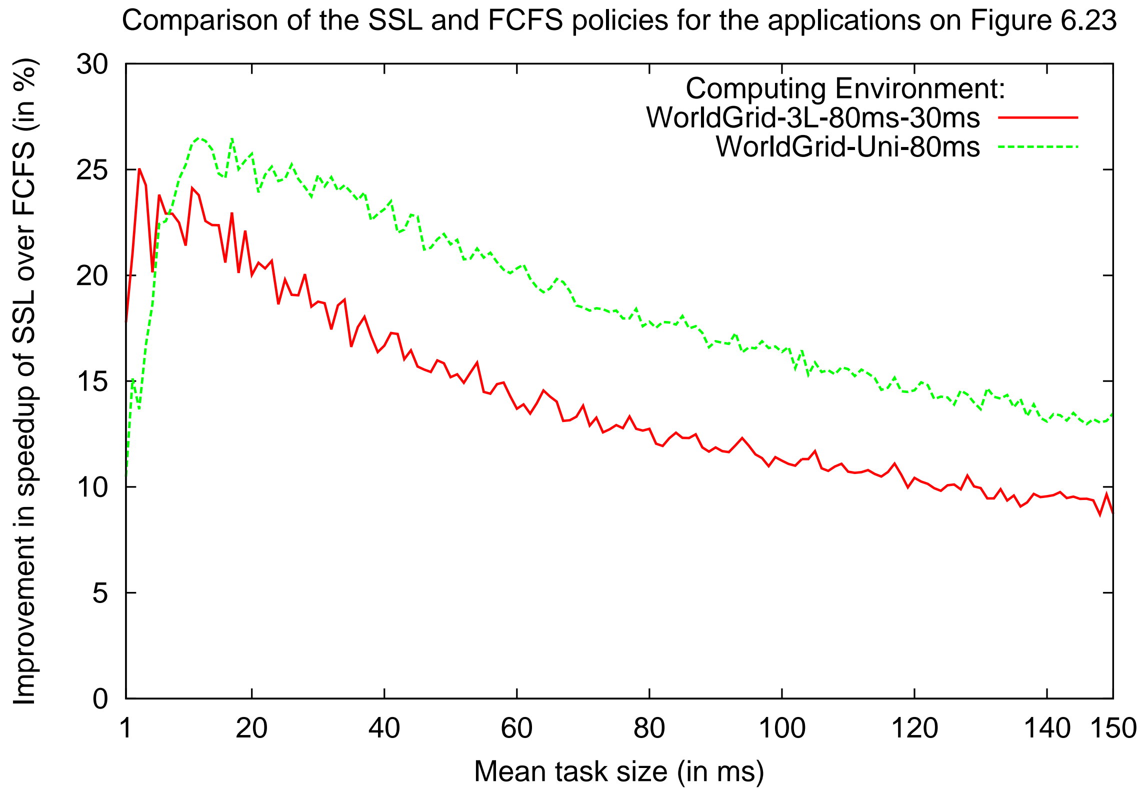


Figure 6.26: Improvements that SLL brings over FCFS for the SingleDataPar(4800,m,0.9) applications on the WorldGrid-3L-80ms-30ms and WorldGrid-Uni-80ms environments

lection policies. We can see that on Figure 6.26, which compares the improvements of the SSL policy over FCFS for the WorldGrid-3L-80ms-30ms and WorldGrid-Uni-80ms (the environment that consists of 8 clusters, where the latency between all clusters is 80ms) environments. Although the WorldGrid-3L-80ms-30ms environment is more heterogeneous, we can observe better improvements on the WorldGrid-Uni-80ms environment, since communication latencies between the main cluster and other clusters are higher there.

Very similar results can be obtained when we consider the applications with the fixed mean task size and the degree of irregularity, and variable number of tasks. For example, Figure 6.27 shows the speedup improvements that the SSL policy brings over FCFS for the SingleDataPar(t,30ms,0.9) applications on the WorldGrid environments. We can observe that for the applications that comprise a smaller number of tasks, the best improvements are obtained for more homogeneous environments. This is somewhat expected, since for more heterogeneous environments with higher communication

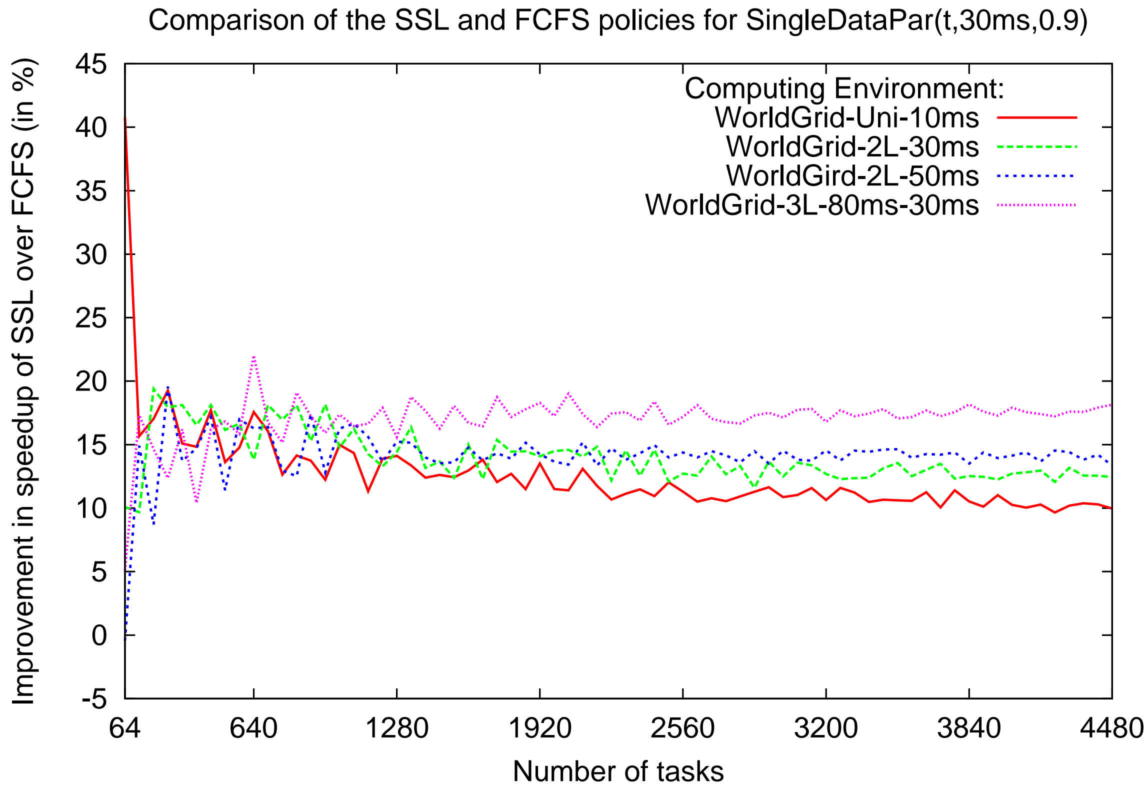


Figure 6.27: Improvements that SLL brings over FCFS for SingleDataPar(t,30ms,0.9) applications on WorldGrid environments

latencies, the applications' tasks are too fine grained (relative to the communication latencies), and this resembles the case of applications with a small number of fine grained tasks (considered in Section 6.3.3, Figure 6.10), where we have seen that the SSL policy does not bring notable improvements to the FCFS one. However, as the number of tasks in the application increases, we can clearly observe that the improvements are better on more heterogeneous environments with higher communication latencies than on more homogeneous environments.

From these experiments, we can conclude that the granularity-driven task selection policies give better speedup improvements on larger computing environments, where the communication latencies between the clusters are higher.

Experiments from this section show that the SSL policy is also the best granularity-driven task selection policy for the computing environments with multiple levels of communication latencies. We may also want to consider the refinements of the SSL policy for these environments. Instead of just one SSL policy, which sends small tasks to thieves from the same cluster, and large tasks to thieves from the remote clusters,

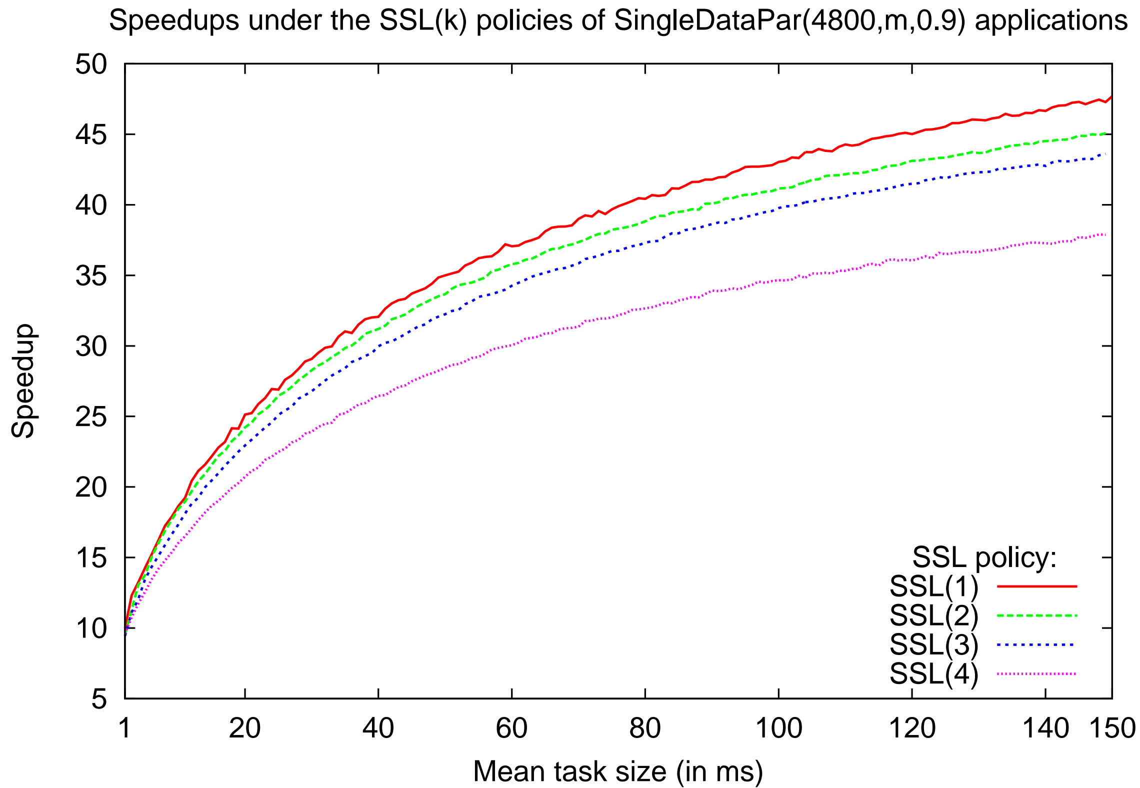


Figure 6.28: The performance of the SSL(k) policies for SingleDataPar(4800,m,0.9) applications on WorldGrid-3L-80ms-30ms computing environment

we can consider the SSL(k) policies, where k is a distance between a victim and a thief (in terms of communication latency levels) after which large tasks are sent as responses to the thieves steal attempts. If the distance between the victim and the thief is less than k, the smallest task from the victim's task pool is sent to the thief.

For example, in WorldGrid-3L-80ms-30ms, for each victim there are 5 different levels of communication latency to which thieves can belong: same-PE (level 0, with latency 0ms), same-cluster (level 1, with latency 0.1ms), same-country-cluster (level 2, with latency 10ms), same-continent-cluster (level 3, with latency 30ms) and different-continent-cluster (level 4, with latency 80ms). SSL(0) corresponds to the LLL policy and SSL(1) to SSL. Under the SSL(2) policy, the victim would send the smallest task to a thief that belongs to the same cluster, or to the cluster which is in the same country, and the largest task to thieves that belong to clusters from different countries or continents.

The question we can ask is can we get better speedups than under the SSL(1) policy if we consider the SSL(2), SSL(3) or SSL(4) policies. Note that SSL(4) corresponds

to SSS policy, which we do not consider in this thesis.

Figure 6.28 gives us some insight into the performance of the various SSL(k) policies. It compares the speedups obtained for the applications considered on the Figure 6.24 on the WorldGrid-3L-80ms-30ms computing environment, under the SSL(1), SSL(2), SSL(3) and SSL(4) policies. We can see that the SSL(1) policy clearly outperforms all other SSL policies, and that, as we increase the level after which large tasks are sent to thieves, we get worse speedups. We can, therefore, conclude that the SSL(1) policy is indeed the best policy, and that the small tasks from the victim's task queue should be reserved only for very near (in terms of communication latency) PEs.

From the experiments presented in this section, we can make the following conclusions:

1. The granularity-driven task selection policies perform well for the larger environments with heterogeneous latencies between clusters (e.g. the WorldGrid-3L-80ms-30ms environment). The relations between the performance of individual task selection policies is very similar on the environments with the heterogeneous latencies between clusters as is on the environments where these latencies are uniform.

2. Better improvements under the granularity-driven task selection policies can be obtained on the environments with the higher latencies between clusters. The cost of transferring tasks between clusters is much higher on these environments, and therefore we get bigger benefits of using the information about task sizes in order to make good decisions about what tasks to send where.

3. The SSL policy is the best one for almost all of the applications that we considered. Furthermore, the extensions to the basic SSL policy for more heterogeneous environments (SSL(k) policies, which consider different "distance" between a victim and a thief after which large tasks are sent as a response to steal attempts) do not bring any improvements to the basic SSL policy. Therefore, the SSL policy is the best one to use on the environments with heterogeneous latencies between clusters.

6.3.6 Applications with nested-parallel tasks

So far we have only considered the applications with a single level of parallelism. These applications comprise a main task, which generates a set of sequential subtasks. However, there is no reason why we could not employ the same reasoning behind the

granularity-driven task selection policies also to the applications with nested parallelism, where some (or all) of the tasks generated by the main task are nested-parallel.

Recall that the task size in the case of nested-parallel tasks is the sum of the sizes of all of its subtasks. This mean that large tasks are also likely to generate more subtasks, so, if we decide to send large tasks to the PEs from remote clusters, it means that we are also likely to send to them the tasks that generate more parallelism. In addition to the benefit of keeping a single remote thief, to which such task is sent, busy for longer time, this also has an added benefit in the fact that more PEs from the thief's cluster will be able to obtain work locally (since the thief itself will create additional tasks). This, therefore, means that, if we assume that Cluster-Aware Random Stealing is used as a base work-stealing algorithm, these PEs will not need to wait for the completion of the remote steal attempts in order to obtain work. Therefore, we expected that reserving larger tasks to the PEs from remote clusters will have even bigger impact on the speedups of the applications with nested parallel tasks.

In this section, we investigate the speedup improvements that the granularity-driven task selection policies bring to the various DCFixedPar applications (most of which we already considered in Section 5.2). Recall that in the DCFixedPar(n,k,C_{seq}, D, C, l) application, each nested-parallel task (including the main task) creates n subtasks. Of these subtasks, each k-th is itself nested-parallel, and the others are sequential. After level l in the task tree of application is reached, all tasks generated by the nested-parallel tasks are sequential. As in Section 5.2, we focus on the applications where each nested-parallel task has trivial 'divide' and 'conquer' phases, and where the size of each sequential task is $5ms$. We will, therefore, abbreviate by DCFixedPar(n,k,l) the application that corresponds to DCFixedPar(n,k,5ms,0.1ms,0.1ms,l).

Considering different DCFixedPar applications enables us to observe the performance of the granularity-driven task selection policies for the applications that have higher and lower density of nested-parallel tasks (and, therefore, higher and lower degree of irregularity).

For our first experiment (Figure 6.30), we considered the DCFixedPar(40,k,4) applications on the WorldGrid-3L-80ms-30ms computing environment, where k ranges from 3 to 11. These applications generate a large number of parallel tasks, and considering different values of k enables us to observe the performance of granularity-driven task selection policies as the number of nested-parallel tasks gets sparser and, therefore, higher the degree of irregularity of an application is. See Table 6.29 for the number of tasks and the degree of irregularity of each of the DCFixedPar(40,k,4) application we considered. Note that for $k = 7$ and $k = 8$ (and, likewise, for $k = 9$ and $k = 10$) we

k	Number of tasks	The degree of irregularity
3	1237640	0.103284
4	444440	0.158401
5	187240	0.224318
6	62200	0.345560
7	31240	0.450674
9	13640	0.618618
11	4840	0.917055

Figure 6.29: The number of parallel tasks and the degree of irregularity of the considered DCFixedPar(40,k,4) applications

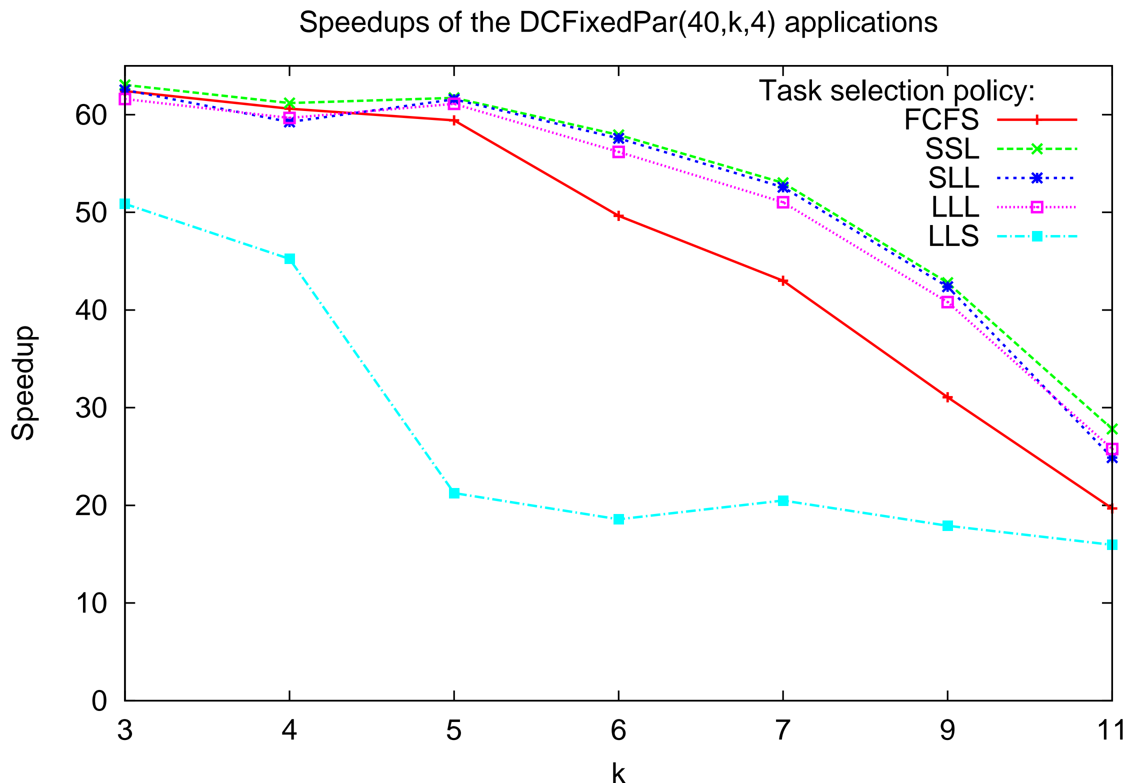


Figure 6.30: Speedups of the DCFixedPar(40,k,4) applications on the WorldGrid-3L-80ms-30ms computing environment

get an identical application, therefore we omit the cases where $k = 8$ and $k = 10$ from the Figure 6.30.

From Figure 6.30, we can observe that for smaller values of k (i.e. for more regular applications) all of the task selection policies perform similarly, giving the same speedups. However, as k increases (and, therefore, the applications become more ir-

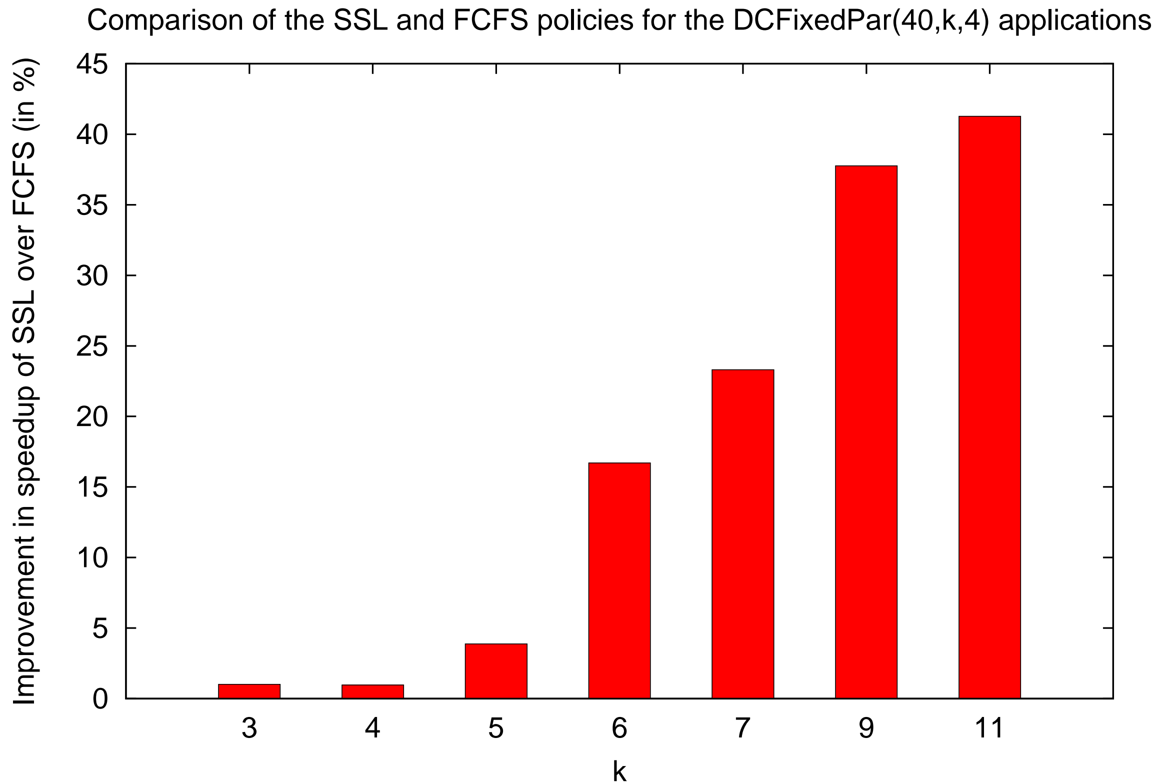


Figure 6.31: Improvements that the SSL policy brings over FCFS for the DCFixed-Par(40,k,4) applications on the WorldGrid-3L-80ms-30ms computing environment

regular), we can observe that the granularity-driven task selection policies start to notably outperform the FCFS policy. We can also observe that all of the granularity-driven task selection policies (except, again, for the LLS policy, which gives the worst speedups of all considered policies) give about the same speedup. The SSL policy is very slightly better than the SLL, and SLL is slightly better than LLL. All of these policies give a very good speedups. This indicates that, for the applications with nested-parallel tasks, the crucial decision is to reserve large tasks to the thieves from remote clusters, and additionally reserving small tasks for local thieves brings only a marginal additional benefit in speedup.

Figure 6.31 shows the speedup improvements that the SSL policy brings over FCFS for the applications considered on Figure 6.30. We can observe a very good improvements (up to 42%) for more irregular applications. Furthermore, we can observe that, similarly to the applications with only a single level of parallelism we considered in the previous sections, the improvements under the SSL policy are increasing as the application's irregularity increases.

k	Number of tasks	The degree of irregularity
15	155500	0.575651
20	78100	0.744058
25	34100	1.012648

Figure 6.32: The number of parallel tasks and the degree of irregularity of the considered DCFixedPar(100,k,4) applications

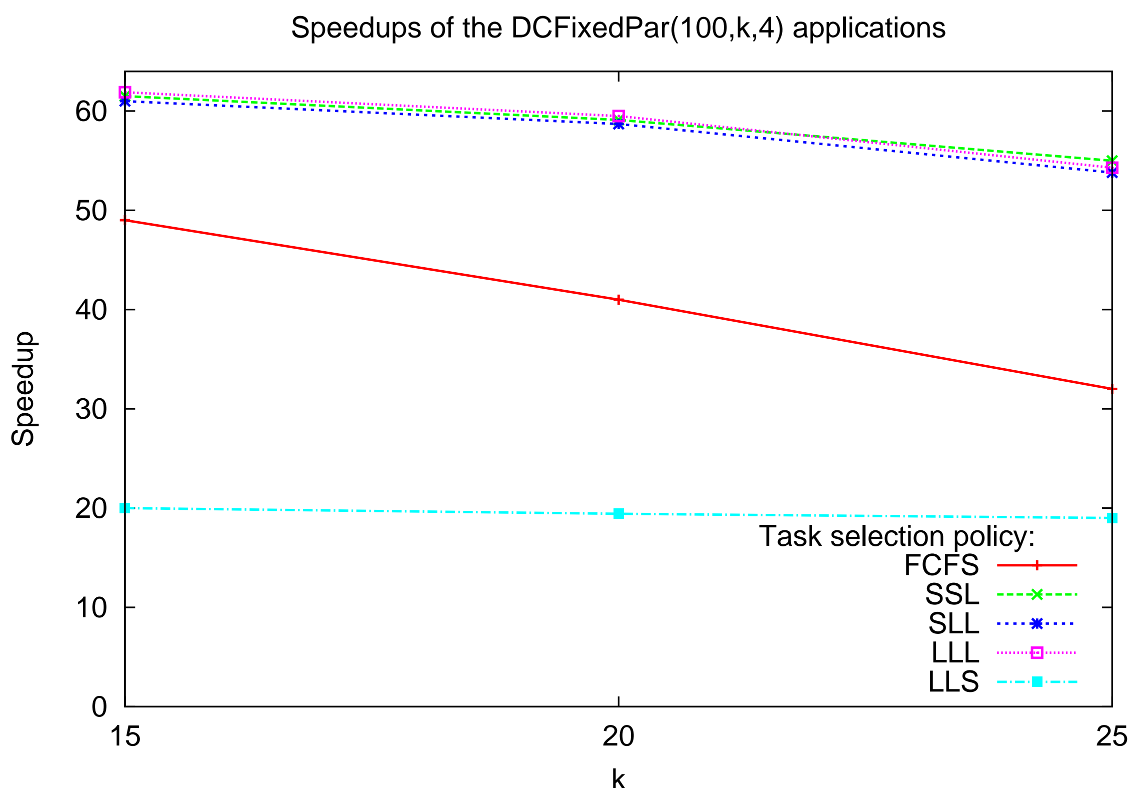


Figure 6.33: Speedups of the DCFixedPar(100,k,4) applications on the WorldGrid-3L-80ms-30ms computing environment

Next, we consider the applications where each nested-parallel tasks generates more subtasks than in the case of the DCFixedPar(40,k,4) applications, but where the nested-parallel tasks are sparser. As an example of this kind of applications, we will consider the DCFixedPar(100,k,4) applications, where k takes values $15, 20$ and 25. The number of tasks and the degree of irregularity of each of the applications of this kind is given in Figure 6.32. Figure 6.34 shows the speedups of these applications under all of the task selection policies that we consider. We can observe similar results as on Figure 6.30 – all of the granularity-driven task selection policies (except for LLS)

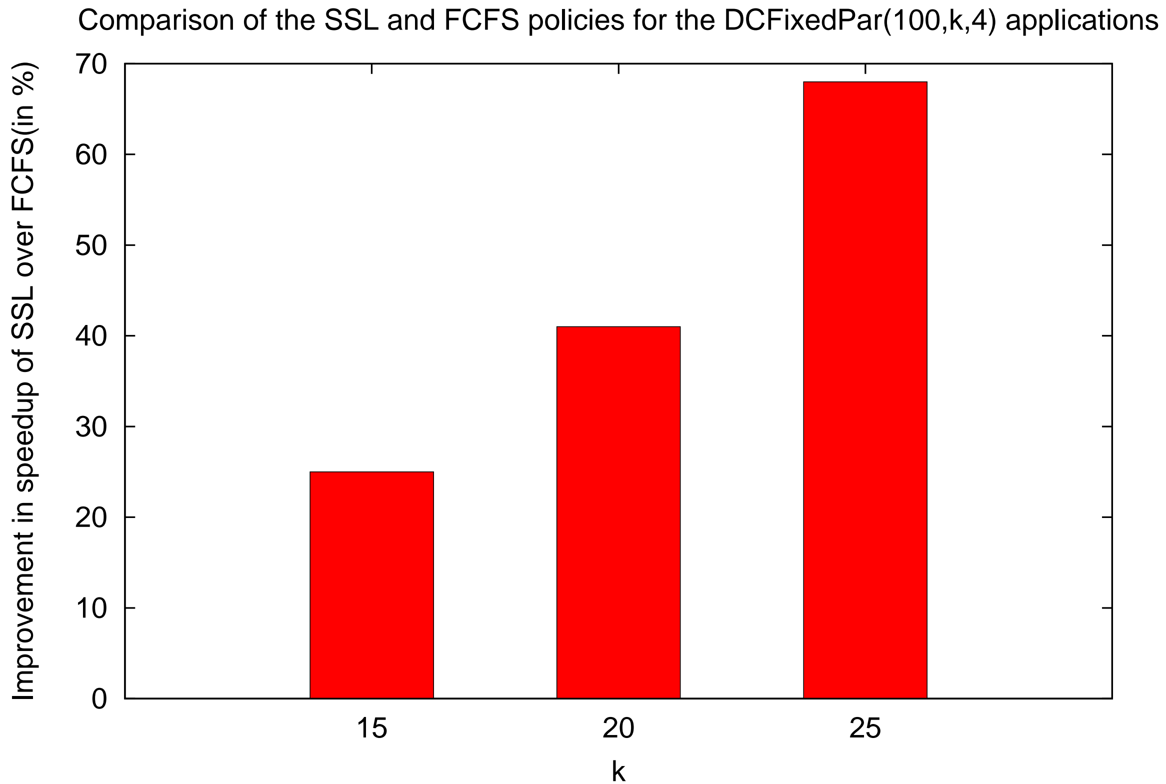


Figure 6.34: Speedup improvements that the SSL policy brings over FCFS for the DCFixedPar(100,k,4) applications on the WorldGrid-3L-80ms-30ms computing environment

give a very similar speedups, and they all notably outperform the FCFS policy. The SSL policy is again very slightly better than SLL and LLL, but the difference between all three policies is minimal.

Figure 6.34 shows the improvements that the SSL policy brings over FCFS for the applications considered on Figure 6.33. Again, we can observe that the improvements are increasing from 22% up to 68% as the value of k increases from 15 to 25.

From the experiments with the applications with nested-parallel tasks, we can conclude the following facts:

1. The granularity-driven task selection policies can significantly improve the speedups of the applications that have nested-parallel tasks, provided that they are irregular.

2. All of the granularity-driven task selection policies (except for the LLS one) give very similar speedups. This indicates that the crucial decision for all policies

is to reserve the large tasks for the thieves from remote clusters, as this is the common denominator for SSL, SLL and LLL policies.

3. The amount of the speedup improvements that these policies bring is bigger than for the applications with only a single level of parallelism. We have observed the improvements of up to 68% for irregular DCFixedPar applications, and the typical improvements for irregular applications are above 20%.

4. The speedup improvements that the granularity-driven task selection policies bring to the FCFS one are better for more irregular applications with nested-parallel tasks.

6.3.7 Where do the Improvements Come From?

In Sections 6.3.2 – 6.3.6, we have observed that the granularity-driven task selection policies bring improvements to the speedups for most of the applications with a high degree of irregularity. Our hypothesis was that the policies that send large chunks of works to more remote thieves will manage to keep these thieves busy for more time, and, consequently, they will have to look for work less often. To see that this is really the reason for the improved speedups of irregular applications, we now consider the execution of an example SingleDataPar(1000,30ms,0.9) application on the Grid(2,8,30ms,0.1ms) environment. Figure 6.35 shows the utilisation of all PEs in the environment. That is, it shows how much of the total execution time are PEs spending in executing tasks. The PE utilisation is on the scale from 0.0 (meaning that a PE spends no time in executing tasks) to 1.0 (meaning that a PE spends all time in task execution). We compare the PE utilisation under the SSL and FCFS policies. All parallelism is created on PE 1 (the main PE), PEs 2-8 belong to the main cluster and PEs 9-16 belong to the remote cluster. We can see that the utilisation of the PEs from the main cluster is about the same for SSL and FCFS policies. However, the utilisation of the PE from the remote cluster is increased from between 0.3 and 0.4 (under FCFS) to between 0.5 and 0.6 (under SSL). This is just what we predicted, as the PEs from the remote cluster spend less time in looking for work and more time in task execution under the SSL policy than under the FCFS one.

Figure 6.36 shows the number of tasks executed on each PE in the environment, for the same application, under the FCFS and SSL policies. We can see that under the SSL policy, the PEs from the main cluster execute a larger number of smaller tasks, compared to the FCFS policy, and that the PEs from the remote cluster execute a smaller number of larger tasks. This means that the the PEs from the remote cluster

spend less time in obtaining work, which explains why their utilisation is increased under the SSL policy.

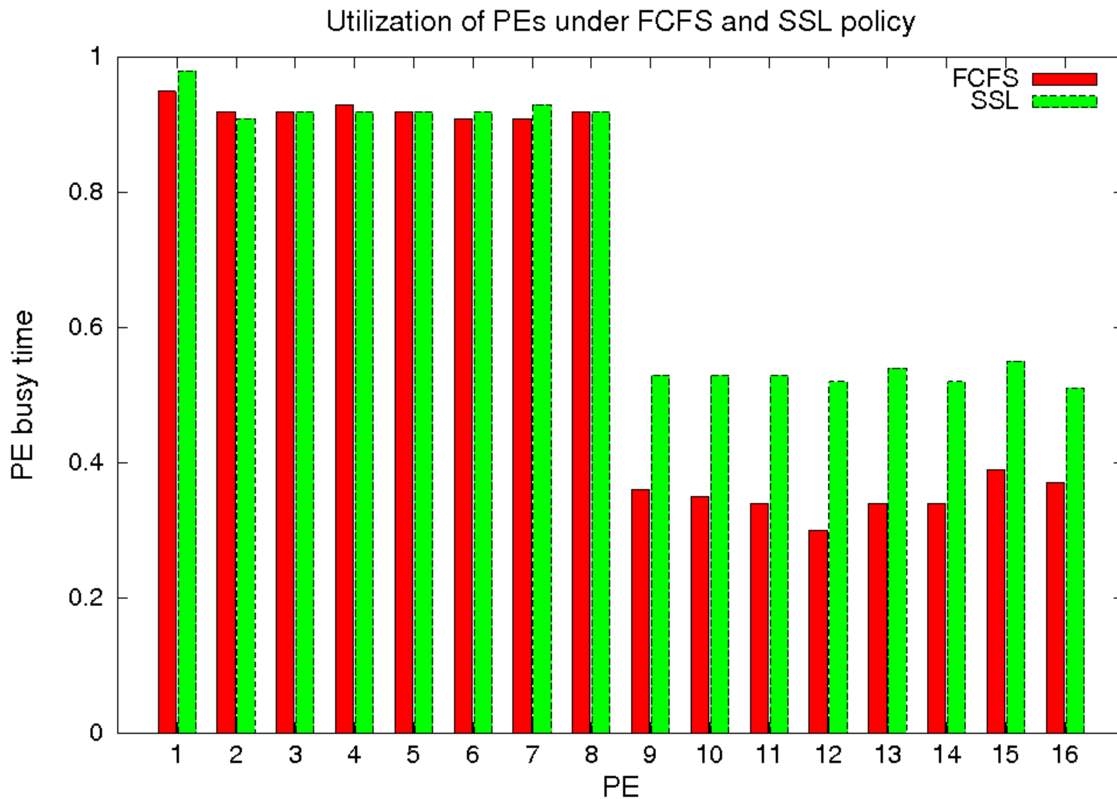


Figure 6.35: Utilisation of PEs under the FCFS and SSL policies for an example SingleDataPar(1000,30ms,0.9) application on the Grid(2,8,30ms,0.1ms) environment.

6.3.8 Summary of the Simulations Experiments

In Sections 6.3.2 – 6.3.7, we have presented a comprehensive evaluation of the granularity-driven task selection policies for a wide class of applications on various heterogeneous computing environments. Taking into account all of the experiments conducted there, and with respect to the hypotheses we described in Section 6.3.1, we can make the following conclusions about the granularity-driven task selection policies:

1. *The granularity-driven task selection policies bring the improvement for most of the parallel applications, compared to the policies that do not use any information about applications' task sizes. The improvements are higher the more irregular the application is.* – For almost all of the applications considered in Sections 6.3.2

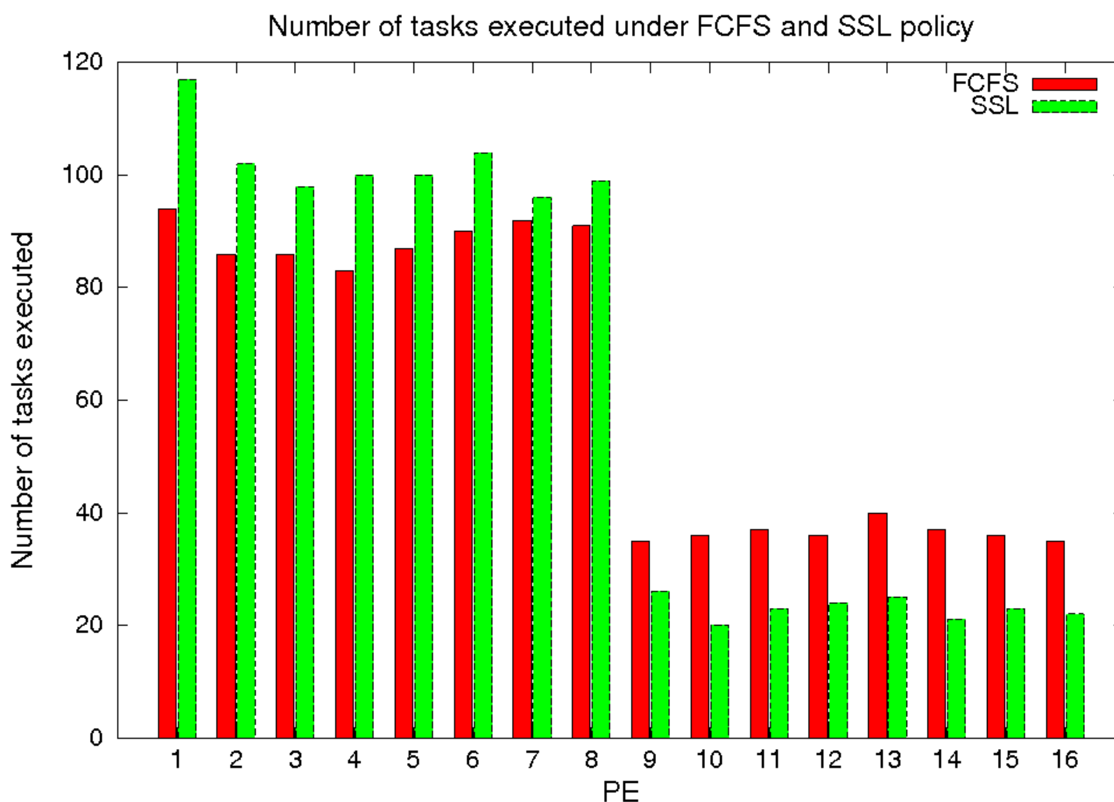


Figure 6.36: Number of tasks executed on PEs under the FCFS and SSL policies for an example SingleDataPar(1000,30ms,0.9) application on the Grid(2,8,30ms,0.1ms) environment.

– 6.3.7, both on the computing environments consisting of only two clusters (Sections 6.3.2 – 6.3.4) and on the larger environments with a hierarchy of latencies (Section 6.3.5), we have noted at least some improvements in the speedups when information about the sizes of tasks that the application comprise is used in work-stealing. The only case where the granularity-driven task selection policies did not bring improvements in speedups, compared to the FCFS policy, is for applications that comprise a very small number of very-fine grained tasks. For these applications, however, the execution times are very small, so any task selection policy is a good choice. Therefore, there are virtually no cases where the use of information about the sizes of application's tasks notably degrades the application's performance.

Furthermore, we have shown that the improvements that the granularity-driven task selection policies bring increase as the applications' degree of irregularity increases, both for the applications with a single level of parallelism (Section

6.3.4) and for the applications with nested parallelism (Section 6.3.6).

2. *The SSL policy gives the best speedups for almost all of the applications* – Both for the single-level data parallel applications that have variable task granularity (Sections 6.3.2 and 6.3.5) and variable number of tasks (Sections 6.3.3 and 6.3.5) and for the applications with nested parallelism (Section 6.3.6), we have observed that the best speedups of all considered task selection policies are obtained under the SSL policy. The only case where this policy did not give the best speedups were the applications that comprise a small number of tasks, and where tasks are either very-fine or very-coarse grained (Section 6.3.3). For these applications, the SLL and LLL policies outperform the SSL one. Therefore, we can conclude that for almost all of the applications, the SSL policy is the one to choose.

3. *The SSL, SLL and LLL policies outperform the LLS one for all of the applications that we have considered* – In almost all of the cases, the performance of the SSL, SLL and LLL policies is similar. The LLS policy performs much worse than all of the other granularity-driven policies, and worse than even the FCFS one in almost all of the cases. This shows that the reasoning to leave the small tasks for the PEs from the remote clusters, which works the best for applications with a very small number of tasks, is not applicable to larger applications.

4. *The granularity-driven task selection policies have more impact on the speedups of the applications with finer grained tasks.* – We have seen in Section 6.3.2 that the speedup improvements that the granularity-driven task selection policies bring to the FCFS one are more notable for the applications with finer-grained tasks. For the applications with coarser-grained tasks, the overheads in transferring the tasks even over high-latency networks are negligible compared to their sequential sizes, so all of the policies give very similar speedups. Still, even for this kind of applications, the granularity driven task selections bring small, but measurable, improvement over the FCFS policy in the applications' speedups.

5. *The reason for the speedup improvements that the SSL, SLL and LLL policies bring to the FCFS one are that these policies are able to keep thieves that are farther away from victims busy for longer time, eliminating their need to look for work more often* – This was shown in Section 6.3.7, for the applications comprising a larger number of tasks, with only a single level of parallelism. We have seen that the utilisation of the PEs closer to the main PE (which generates all of the parallelism) might slightly decrease under the SSL policy, compared to

the FCFS one (the reason for that being that they execute smaller tasks than in the case of FCFS policy). However, this is outweighed by much better utilisation of the PEs from the remote cluster.

6.4 Grid-GUM Implementation

In this part of the chapter, we describe and evaluate the implementation of the granularity-driven task selection policies in the Grid-GUM runtime environment. As a reminder (see Section 2.4.3), Grid-GUM differentiates between *sparks*, which are heap pointers to the closures which represent unevaluated expressions, and *threads*, which are actual units of computation that evaluate expressions in parallel. Sparks are very lightweight, and every PE has its own *spark pool*, where sparks that point to the closures from that PE's heap are kept. When a PE gets idle, it converts one of the sparks from its spark pool into a thread, and starts evaluating it. The steal operation in Grid-GUM involves transferring one or more sparks from the victim's to the thief's spark pool. Relating this to our simulation setup, we can see sparks as tasks whose execution has not yet started, so they can be transferred between PEs, and threads as tasks whose execution has started and that are, therefore, immovable. Task selection policies, therefore, decide on which spark is transferred from the victim's to the thief's spark pool in steal operation.

In order to make our results more generally applicable, we assume that only one task is transferred in each steal operation, as this is the case in most of the parallel runtime systems. Therefore, in our Grid-GUM implementation, we also restrict the number of sparks offloaded in a single steal operation to just one.

In this section, we present the evaluation of the SSL, SLL, LLL and LLS task selection policies for managing spark pools. We have tested these policies on a set of synthetic applications (i.e. artificially generated applications with controllable thread sizes and the number of sparks). The purpose of these experiments was to mimic the applications considered in the simulation setup and in that way to observe the relation between the results obtained using the simulations and the real implementation for the same classes of applications on the same computing environments.

6.4.1 Implementation Details

The information about the sizes of the threads created from the sparks during the application execution is obtained using profiling. For each application, we assign a unique name for each spark that might be created during the application execution.

This name is then used in a runtime system as an identifier of a thread created from that spark. We have profiled the same application over several runs on a parallel machine to record the time each thread spends in computation. This information is then used as an approximation of the size of the thread that will be created from its spark.

Assigning names to sparks is done by using a `parWithName` combinator, which is similar to `par` but, in addition, passes the name of the spark to the runtime system. That is, `parWithName name p q` creates a spark for the expression p, puts it into a spark pool and assigns a name `name` to it. We have used this combinator to extend the `Strategies` module, to include the strategies that assign names for the sparks that they create. For example, in addition to `parList strat` (which applies `strat` strategy to each element of a list in parallel), we also have `parListWNames prefix strat` strategy, which gives a name (starting from `0`, prefixed with `prefix`) to each spark created from an application of a strategy `strat` to a list in parallel[3]. In this way, by using different values for `prefix` for different strategic applications, we can assign a unique name for each spark created in the application, even if it has various (possibly nested) strategic applications.

As an example, the following code has two nested `parMap` applications:

```
res = f [[10,2],[23,6],[15,8]]
where
    f = parMapWNames "top:" g rnf
    g = parMapWNames "bot:" someFun rnf
```

A spark resulting from the application of `someFun 10` (which is sparked as a part of the evaluation of `g [10,2]`) will have the name `top:0bot:0`, `someFun 2` will have name `top:0bot:1`, `someFun 23` will have name `top:1bot:0` and so on.

6.5 Experiments with Grid-GUM

In this section, we evaluate the performance of the granularity-driven task selection policies in Grid-GUM. As we have mentioned before, the main purpose of these experiments is to compare the results obtained in the simulation experiments, which ignored a lot of additional overheads that might be present in realistic runtime systems, with the ones done with the real implementation of the granularity-driven task selection policies in the realistic runtime system. In addition to exploring how do these policies

[3]This, of course, gives rise to various other list strategies (such as parMapWNames) that use parListWNames as its base strategy

perform in realistic settings, this also has the purpose of testing the validity of the results obtained under the simulations, i.e. to observe whether ignoring all overheads except for the ones that come from the communication latencies in the environment gives us a good estimation of the performance of the various task selection policies for work-stealing.

Since Grid-GUM is built on top of ghc 4.06 sequential Haskell compiler, we have used this compiler as a baseline for our results. In other words, sequential runtimes of parallel applications are obtained under ghc 4.06, when all of the `par` and `seq` combinators are removed from the application code.

6.5.1 Differences in Simulation and Grid-GUM Setup

Compared to the simulation setup, where the only overheads in the application execution came from the communication latencies in the underlying computing environment, in Grid-GUM there are some additional overheads that can have notable impact on the applications' speedups. Some of the overheads that we have ignored in the simulations include:

- Time to create (or process) a message in the simulation setup was assumed to be 0 (i.e. negligible). This might not be the case with Grid-GUM. For example, packing the subgraph of a spark into MPI buffers and unpacking it at the destination site can add non-negligible overheads in processing `FISH`, `SCHEDULE`, `FETCH` and `REPLY` messages. In addition to that, there are also overheads in sending each message, due to an internal operation of the implementations of MPI libraries.

- Thread creation (as well as the destruction when the thread finishes evaluation), which involves initialisation/destroying of a TSO structure, adds a small overhead to each thread created during the application execution in Grid-GUM. This overhead was also assumed to be 0 in simulations.

- On-demand data fetching can introduce additional overheads in the transfer of the data needed for the thread evaluation to a remote destination, since more than one message might be needed for this. Depending on the size of the spark's subgraph, the number of thunks in that subgraph and the policy of thunk offloading, not all of the data needed for thread evaluation might be sent in a single `SCHEDULE` message to the destination PE. Additional `FETCH` (and corresponding `REPLY`) messages may be needed to fetch the additional data.

- In addition, due to on-demand data fetching, once the thread finish evaluation, it does not send the result back automatically to the PE where its spark was originally created. In other words, threads do not send their results back to the PEs that hold their parent threads. Instead, the parent thread request this data via the `FETCH` message, and the result is then sent back to it via `REPLY` message. Depending on the size of the result, many `FETCH` and `REPLY` messages need to be exchanged in order to send the whole result back to the parent thread.

- Garbage collection, which can in some cases significantly contribute to the execution time of an application, was completely ignored in simulations.

Due to these overheads, we expected the speedups of applications under Grid-GUM to be to some degree different than of the models of equivalent applications under the simulations.

As we have mentioned before, one additional thing that was ignored in the simulations were the data-dependencies between threads. In GpH, the data-dependencies are synchronisation points between threads. Thus, if a thread needs a value that is being computed by some other thread, it will block (after sending `FETCH` message) and will remain blocked until the value it needs is computed and sent to it. Since the applications that we consider here are embarrassingly parallel, ignoring the overheads in thread synchronisation will not have a significant impact on the speedups of the Grid-GUM applications.

6.5.2 Experiments with the Synthetic Applications

This section describes the experiments and the results that we have obtained with the synthetic applications. These are the artificial applications that we have created to compare how do the improvements that the granularity-driven task selection policies bring for Grid-GUM compare to the ones obtained under simulations in Section 6.3. All experiments were performed on an environment consisting of two 8-core Intel Xeon servers, running CentOS Linux. In order to obtain a heterogeneous communication setup, we have introduced additional message latency between multicore machines using the `netem` kernel module, giving a total latency between different servers of 30ms. The latency between the cores within the same server was 0.1ms. This gives us the identical environment to Grid(2,8,0.1ms,30ms), which was used in most of the simulation experiments.

In order to eliminate the degree of randomness in our results, each application we test was executed 100 times under Grid-GUM on the computing environment we

consider, and the average speedup was taken as representative.

Our focus is on the applications without nested-parallelism, resembling the Single-DataPar applications. These applications consist of a single `parMap` application of a function to a list of numbers. The number of sparks created in the application is equal to the length of a list. The function `f` applied to elements of a list in `parMap` is chosen so that the time to compute `f(x)` sequentially is approximately `x` milliseconds.

In our settings, we have measured that the time it takes for `pfib 20` to be evaluated is approximately 1ms, so the function we have used is

```
f 0 = 0
f x = fib 20 * 0 + f (x-1)
```

In section 6.5.1, we have pointed out to some of the differences in the simulation and the Grid-GUM setup (i.e. additional overheads present in Grid-GUM). It is also worth noting that, having ignored the overheads in task creation and sorting the task pool, in non-nested data-parallel applications under the simulations, the main task creates all child tasks *immediately* at the start of the execution. Therefore, when steal message arrives to the main PE, all parallel tasks are readily available. In Grid-GUM, however, there are certain overheads in creating sparks and, in the case of the granularity-driven task selection policies, in sorting the spark pool each time a new spark is added to it. This means that it takes non-trivial amount of time to actually create all sparks and organise them (according to their sizes) in the spark pool, even if all of them are created right at the beginning of the execution of the main thread. This further means that a `FISH` message may arrive to the main PE before it has created all the sparks. Since scheduling in Grid-GUM is preemptive, the `FISH` message is processed almost immediately, which means that only *currently available* sparks are considered in task selection policies. This means that the granularity-driven task selection policies will choose the smallest or the largest currently available spark, which might not be the largest or the largest spark altogether, as is the case in simulations. Therefore, the decisions of what spark will be sent to what thief might differ for identical applications in the simulations and in Grid-GUM.

Out of all Grid-GUM overheads mentioned in Section 6.5.1, the overhead of the main thread having to collect the results of its sparked subthreads presents the biggest problem for the kind of applications we are considering here. Since the applications consist of large number of tasks, some (or all) of which can be very fine grained, the final phase where only the main thread is active and it is collecting the results can, due to high latencies to the remote PEs, easily take most of the time of application execution. We have measured that, with applications that create thousands of fine-grained sparks,

the phase of collecting the results can take about 75% of total application execution time.

To solve this problem, we have, for the purpose of data-parallel applications, modified the Grid-GUM runtime system so that, upon completion, the result of a thread is automatically sent back to the PE where thread's spark was originally created. That is, if the thread is created from the stolen spark, we remember the spark's top level closure (which is initially `THUNK`, but is eventually updated to a normal form value), and when the thread finishes, we pack the subgraph of this closure and send it back to the original victim. This eliminates the need of a thread which created the spark to collect its result via `FETCH-RESUME` message exchange. Consequently, the long results collecting phase is eliminated.

As we did in the simulation experiments, here we have also considered the applications with different degrees of irregularity, different numbers of tasks and different mean task sizes.

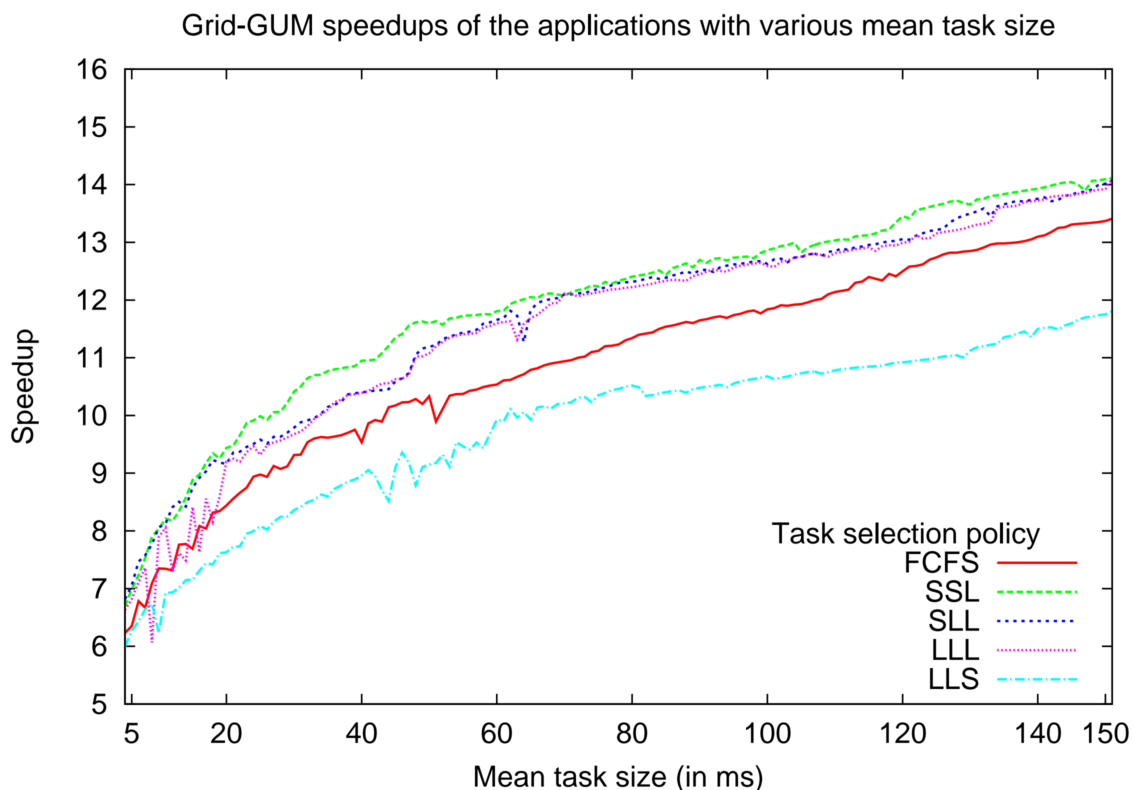


Figure 6.37: Speedups under Grid-GUM for applications with variable mean task size, 1600 tasks and the degree of irregularity of 0.9

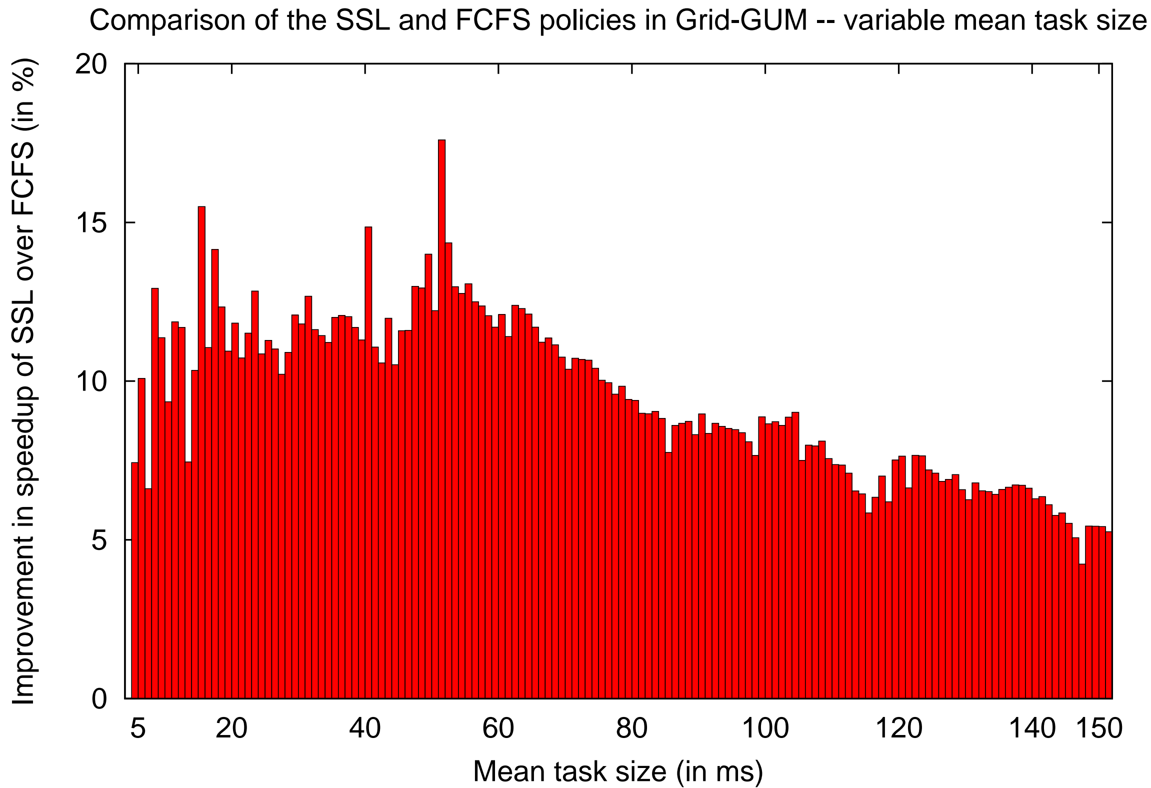


Figure 6.38: Improvements in speedups of SSL over FCFS for applications with variable mean task size, 1600 tasks and the degree of irregularity of 0.9

Applications with a Variable Mean Task Size

Figure 6.37 shows the relative speedups obtained for the applications with 1600 tasks, variable mean task size and the degree of irregularity of 0.9. These applications correspond to the SingleDataPar(1600,m,0.9) applications considered in Section 6.3.2. Therefore, we can compare the results from Figure 6.37 with the results on Figure 6.3. Note that the measurements done here exclude the time needed for system initialisation (which can have substantial impact on absolute speedup in the case of smaller applications).

We can notice similar results Figures 6.37 and 6.3. The SSL policy gives the best speedups for almost all of the applications considered on Figure 6.37; the SSL, SLL and LLL policies outperform FCFS policy, and the FCFS policy outperforms the LLS one. We can, however, also notice some differences between the compared figures. We can see that, under Grid-GUM, the SSL, SLL and LLL policies give very similar speedups for the applications that have coarse-grained tasks.

Figure 6.38 shows the improvements in speedups of the considered applications that

the SSL policy brings over the FCFS one. Again, when we compare the results on this figure with the ones for the similar class of applications under simulations (see Figure 6.4 on page 178), we can observe similar improvements. The speedup improvements are higher for the applications with finer-grained tasks, and they become small (but still measurable) for the applications with coarse-grained tasks.

Applications with variable number of tasks

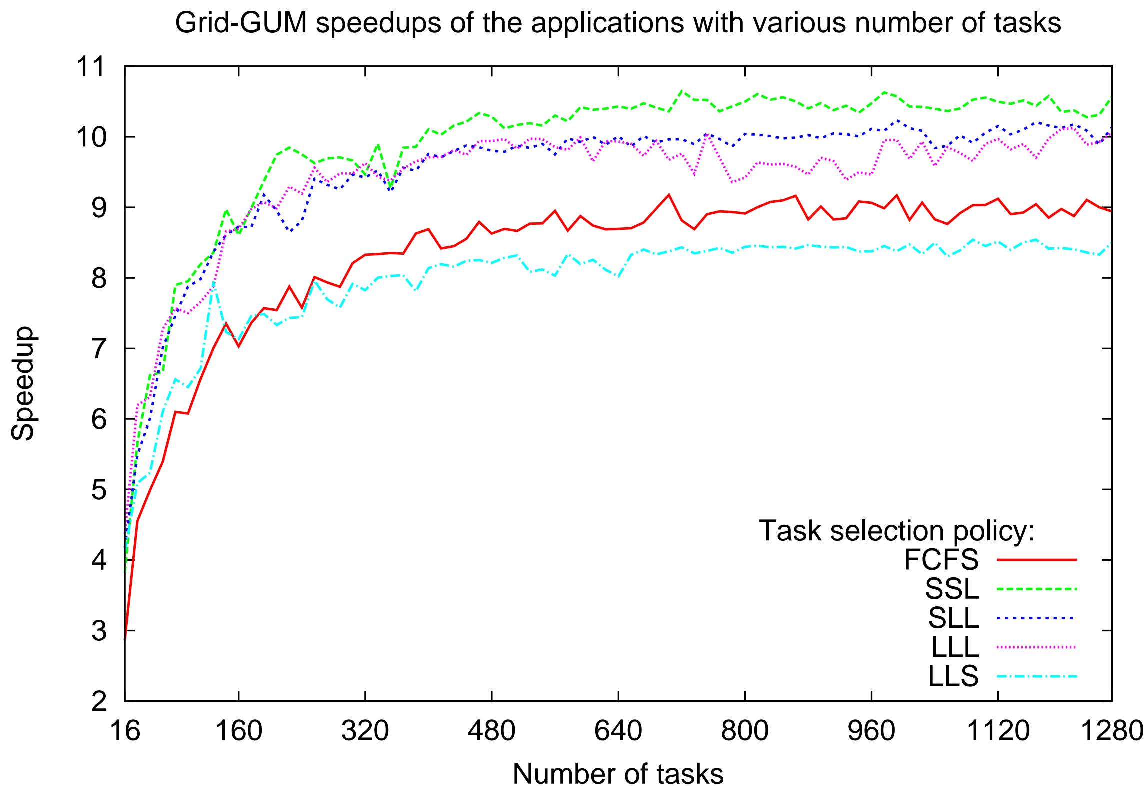


Figure 6.39: Speedups under Grid-GUM for the applications with variable number of tasks, mean task size of 30ms and the degree of irregularity of 0.9

Next, we consider the applications with a variable number of tasks. Experiments done here are equivalent to the ones done in Section 6.3.3. Figure 6.39 shows the speedups under Grid-GUM of applications with the mean task size of 30ms, the degree of irregularity of 0.9 and with a variable number of tasks (the results equivalent to those on Figure 6.13 under simulations, for the SingleDataPar(t,30ms,0.9) applications). When we compare Figures 6.39 and 6.13, we can see generally the same results in terms of the relations between the performance of the granularity-driven task selection policies: the SSL policy gives the best speedups on both figures, granularity-driven

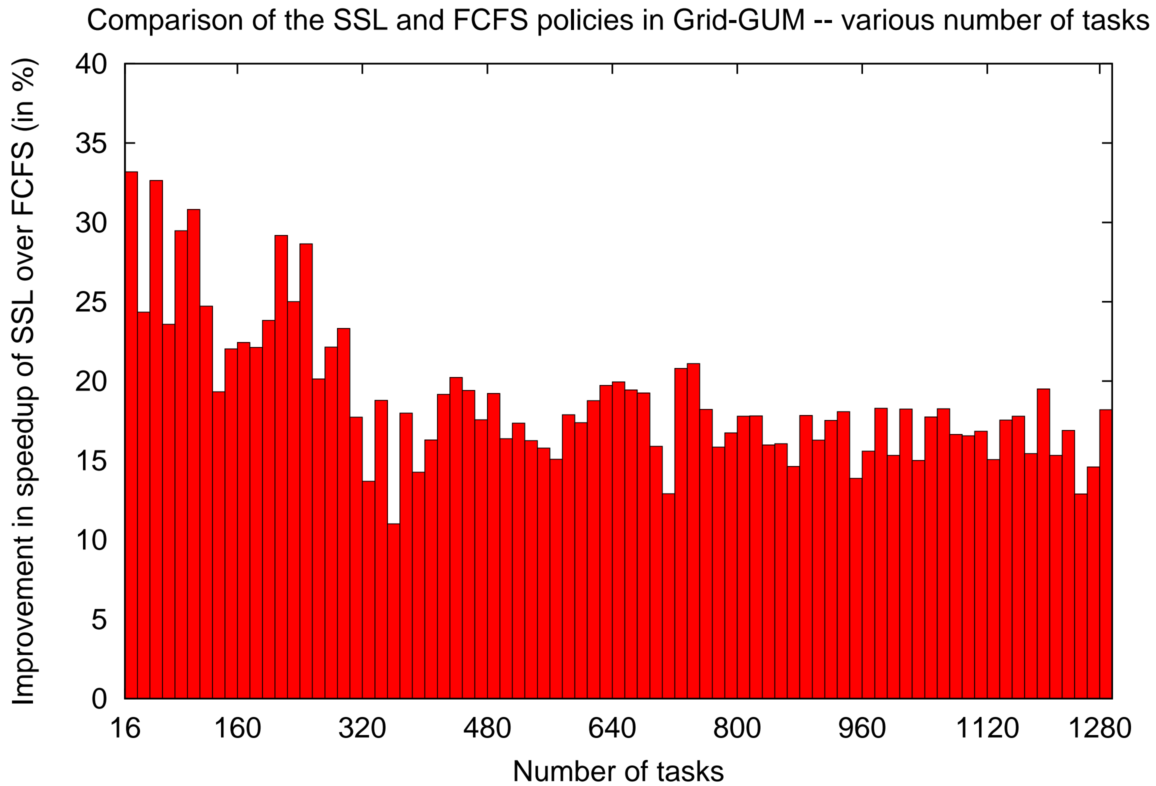


Figure 6.40: Improvements in speedups of SSL over FCFS for applications with mean task size of 30ms, the degree of irregularity of 0.9 and variable number of tasks

task selection policies (except for the LLS one) outperform the FCFS one, and LLS is the worst policy. We can also see that the performance of SSL and LLL policies is pretty similar on both figures. However, one thing that we can notice is that the differences in the speedups between the SSL policy on one side and the SLL and LLL policies on the other are smaller on Figure 6.39 than on Figure 6.13.

Figure 6.40 shows the speedup improvements that the SSL policy brings over the FCFS one under Grid-GUM, for the applications considered on Figure 6.39. The improvements are in the range between 10% and 20% most of the time, and between 20% and 35% for the applications with smaller number of tasks. We can observe that these improvements are better than in the case of simulations (see Figure 6.14 on page 191 for the equivalent simulations results).

The small differences between the simulations and the Grid-GUM implementation results can be explained by the fact that various overheads in task creation/offloading present in Grid-GUM (see Section 6.5.1), that we have ignored in simulation, have the most impact when the tasks are offloaded remotely, over high-latency networks.

Therefore, since the overheads in sending tasks to the PEs from remote clusters are higher in Grid-GUM than in the case of simulations, the improvements that we get by reducing the number of tasks executed remotely (under the SSL, SLL and LLL policies) are better than in simulations. Also, since the main source of improvement is reserving *large* tasks for remote offloading, the additional benefit of saving *the largest* ones (i.e. the benefit of SSL over SLL and LLL) is not that big.

Applications with variable degree of irregularity

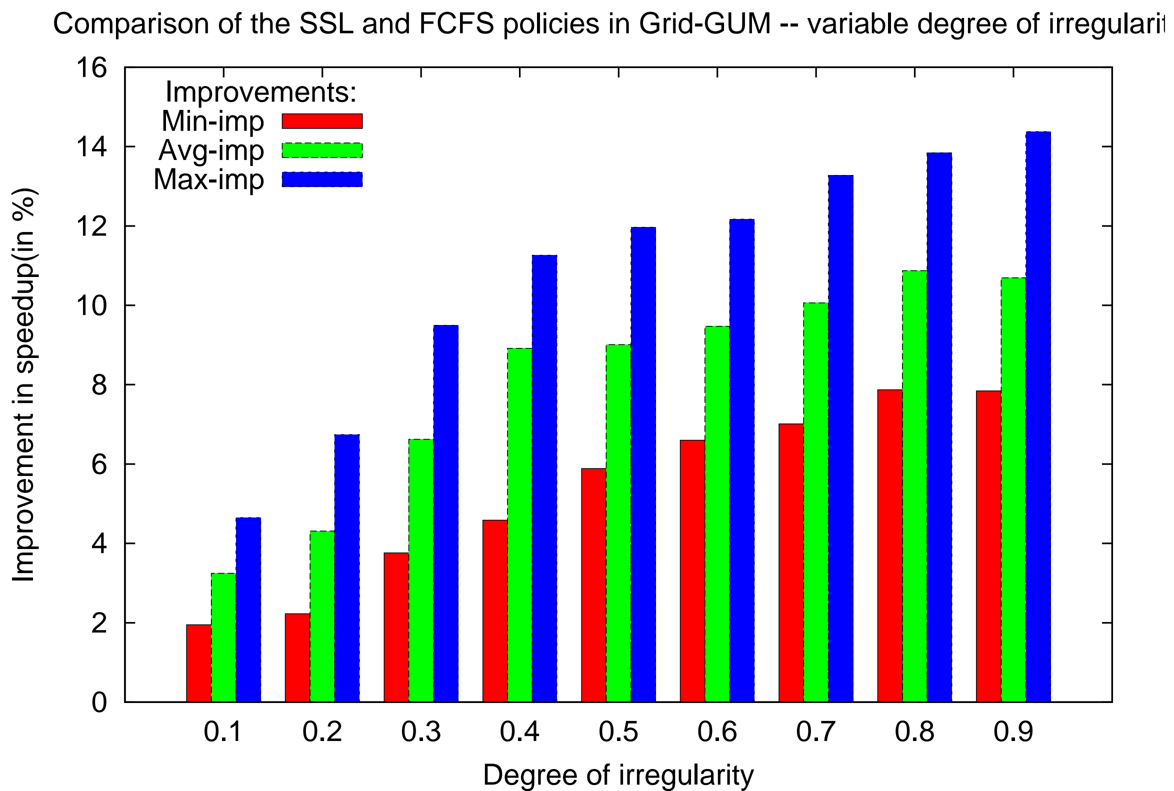


Figure 6.41: Improvements in speedups of SSL over FCFS for applications with mean task size of 30ms, 1600 tasks and variable degree of irregularity

Figure 6.41 shows the improvements that SSL policy brings over the FCFS for applications with mean task size of 30ms, comprising 1600 tasks and having a variable degree of irregularity. Similarly to the experiments from Section 6.3.4, we can observe that minimal, average and maximal improvements are increasing as the applications' degree of irregularity is increasing. Therefore, it is also the case for Grid-GUM implementation that the granularity-driven task selection policies give better improvements for more irregular parallel applications. However, comparing Figure 6.41 and Figure

6.20, we can observe that the differences between minimal, average and maximal improvements are higher in the case of Grid-GUM implementation, than in the case of the simulations.

Despite some differences in the performance of the granularity-driven task selection policies under the Grid-GUM implementation and under simulations, which are a consequence of some additional overheads present in Grid-GUM, we can conclude that our simulations give a very good estimation of what granularity-drive task selection policy works the best for what applications, and of the improvements that we can get with the use of these policies when they are implemented in realistic runtime systems.

6.6 Conclusions

In this chapter, we proposed four simple policies that use information about the task sizes of an application in making decisions about what task to send as a response to a steal attempt. We demonstrated that these policies can bring improvements to the speedups of irregular parallel applications executed on heterogeneous computing environments. The policies considered in this chapter distinguish between three levels of communication latency in underlying computing environment (same-PE, local and remote levels) and they do not rely on the precise knowledge of application task sizes (in terms of, for example, the number of clock cycles) They only assume that it is possible to compare two tasks, and to decide which one of them is larger (in terms of its sequential size, as defined in Section 4.2.1.

We have considered four granularity-driven task selection policies: SSL (Small-Small-Large), SLL (Small-Large-Large), LLL (Large-Large-Large) and LLS (Large-Large-Small), which choose the smallest (or the largest) available task to send as a response to a steal attempt, depending on the latency level of the thief. Each of these policies has a different strategy for improving the applications' speedups:

- **SSL** - reserve all the largest tasks for execution on remote clusters, in order to send more work less often to remote PEs
- **SLL** - only send large tasks and execute all of the small ones on the PE where they are created. We assume that it is only worth sending large chunks of work, and all other work should be executed where it is created.
- **LLL** - greedy approach. We try to execute the largest task as soon as possible.
- **LLS** - execute the large tasks on the PE where they are created, or within the same cluster. By sending only the small tasks, we hope that they will be executed

quickly and that their results will be sent back before the large tasks are executed within the cluster they were created on.

To evaluate the effectiveness of task selection policies, we have measured the improvements that they bring to the speedups of parallel applications, compared to the simple FCFS policy that does not need any granularity information. This policy chooses the tasks for offloading in FCFS manner, and tasks for local execution in LCFS manner. Our main focus was on the applications with only a single-level of parallelism (SingleDataPar applications), but we also considered the examples of applications with nested parallelism (DCFixedPar applications). SingleDataPar applications we considered differed in the number of tasks they comprise, their mean task size and the degree of irregularity, and DCFixedPar ones differed in the number of tasks generated and the degree of irregularity. The granularity-driven task selection policies were tested using both simulations and their implementation in the Grid-GUM runtime system.

No single policy performs the best for all applications, but SSL policy is the best one in vast majority of cases. Only for applications comprising a smaller number of tasks, and whose tasks are coarse grained does the SLL policy outperform SSL. The improvements in speedups obtained with the usage of granularity-driven task selection policies (compared to the basic FCFS one) are in the range of 10-30% most of the time for applications with a single-level of parallelism. For the applications with nested-parallelism, we have observed even better improvements (up to 68% for more irregular applications). We have also observed that the improvements are better the more irregular parallel applications are (i.e. higher the degree of irregularity they have).

The results from simulations were confirmed when synthetic Grid-GUM applications with similar characteristics were considered on the example heterogeneous computing environment. This also proved the practical applicability of the granularity-driven task selection policies.

Chapter 7

Conclusions

The main goal of this thesis has been to improve the state-of-the-art in work-stealing algorithms, in order to obtain good speedups for irregular parallel applications on distributed heterogeneous computing environments. Our approach has been to focus on two main questions:

- Where should thieves look for work during the application execution? Good methods for selecting the stealing targets would enable thieves to obtain work quickly once they are idle, which would contribute to the good PE utilisation and, consequently, good application speedups.

- How should victims respond to steal attempts? Sending the appropriate chunks of work to the thieves would contribute to the better balance of work across the computing environment, avoiding situations where a small set of PEs grabs most of the work.

We can say that the goal of the thesis has been achieved, with some limitations on the kinds of applications and computing environments that we have considered. We have developed the algorithms that are especially tailored to provide good answers to the two posed questions for highly irregular applications. We have shown that significant improvements in speedups can be achieved with our algorithms for irregular applications on heterogeneous distributed computing environments, compared to the current state-of-the-art work-stealing algorithms. Furthermore, we have shown that speedup improvements are larger the more irregular the application and the underlying computing environment are. We have also shown that our algorithms do not bring notable decrease in the performance when used for regular applications or on homogeneous computing environments. Only a few cases exist where the speedups under our

algorithms are slightly worse than the ones obtained under the state-of-the-art ones, but the decrease in speedups in these cases showed to be only minor.

Where should thieves look for work during the application execution?

In Chapter 5 we focused on the first of the two posed questions. Our approach in tackling the problem of locating the stealing targets that have work was to use information about the dynamic PE loads, as well as static information about the network topology. We first investigated methods for selecting the stealing targets, under the assumption that *perfect* load information is available. We have discovered that the *Perfect Cluster-aware Random Stealing* (Perfect CRS) algorithm gives the best speedups for all considered applications on all considered computing environments. This algorithm uses a method for target selection from the Cluster-aware Random Stealing (CRS) algorithm, where local (i.e. within a cluster) and remote (i.e. outside the cluster) steals are attempted in parallel. Perfect CRS differs from CRS in that a target is randomly selected from the set of PEs that have work, and that are inside (for local stealing) or outside (for remote stealing) of the thief's cluster, as opposed to randomly selecting a target from the set of *all* PEs inside/outside of the thief's cluster in CRS.

We have subsequently focused on the setting where perfect load information is not available and where some approximation of load information needs to be obtained. For this purpose, we have developed a novel Feudal Stealing algorithm, which uses CRS as its base. Feudal Stealing extends CRS with a method of exchanging load information between PEs, such that each PE either has a good approximation of the load of all clusters in the environment, or, alternatively, it knows what nearby PE holds this approximation. We have showed that, for highly irregular applications, Feudal Stealing gives speedups that are very close to those under Perfect CRS, in which perfect load information is used.

Chapter 5, therefore, provided the accurate method for estimating dynamic PE loads during the application execution and showed that the best way to take advantage of accurate load information is to do local and remote stealing in parallel.

How Should Victims Respond to Steal Attempts?

In Chapter 6 we focused on the second question posed. We have developed four granularity-driven task selection policies that a victim can use to decide which task to send to a thief, based on the latency of the communication link to the thief and the sizes of tasks from the victim's task pool. We showed that our policies bring improvements in speedups for irregular applications on heterogeneous computing en-

vironments, compared to the ad-hoc policies (which select tasks based on their age) that are used in the current state-of-the-art work-stealing algorithms.

7.1 Contributions of the Thesis

The contributions made in this thesis are as follows:

A novel ***Feudal*** Stealing algorithm for heterogeneous distributed computing environments

We have developed a novel Feudal Stealing algorithm, which uses the CRS algorithm as its base. In addition to the basic mechanism for selecting stealing targets taken from CRS, Feudal Stealing uses a combination of locally-centralised and remotely-distributed information propagation mechanisms to obtain a good approximation of the load of the computing environment. In each cluster in the environment, one PE is nominated as a *head PE*, and this PE holds information about the load of the whole cluster. Cluster load information between head PEs is exchanged in a fully-distributed way. This enables the head PEs to have fully accurate information about the loads of the PEs from their clusters, as well as a good approximation of the loads of the other clusters. Due to this, the head PEs are able to appropriately route the steal attempts that go outside or come inside of their clusters.

We have shown that, for irregular parallel applications, Feudal Stealing outperforms both the CRS algorithm (which is considered the best of all the work-stealing algorithms used in current runtime systems) and the Grid-GUM algorithm, which uses a fully distributed information propagation mechanism. We have also shown that the speedups obtained under Feudal Stealing are very close to the ones obtained under the CRS algorithm with perfect load information. This shows a high accuracy of the dynamic load information obtained under Feudal Stealing.

Novel policies for selecting the tasks for offloading during the work-stealing

We have developed the novel *granularity-driven* task selection policies, which use information about the sizes of tasks from the victims' task queues, together with the information about network topology of the underlying computing environment, to decide what tasks to send to thieves. We have showed that our task selection policies can significantly improve the speedups of parallel applications, compared to the ad-hoc policies currently used in work-stealing algorithms, which select the tasks according to their age, and without any knowledge of their parallel profiles.

Comprehensive evaluation of all of our work-stealing algorithms and policies using simulations

Using simulations, we have conducted a comprehensive evaluation of the algorithms and policies proposed in this thesis on a wide class of parallel applications and computing environments. This has enabled us to show that the algorithms and policies we proposed indeed give good improvements in speedups over the state-of-the-art work stealing algorithms for irregular parallel applications. Furthermore, we have showed that speedup improvements of our algorithms and policies over the state-of-the-art algorithms are larger the more irregular the applications are. Finally, we have shown that the algorithms and policies tailored to the irregular applications on heterogeneous computing environments also perform good for regular applications on homogeneous computing environments, which means that there are virtually no situations where their use can decrease the applications' speedups.

Implementation of granularity-driven task selection policies in Grid-GUM

We have implemented the granularity-driven task selection policies in the Grid-GUM runtime environment for Glasgow Parallel Haskell, and we have conducted the evaluation of the performance of this implementation for a wide class of parallel applications. The benefits of this were twofold. Firstly, we have shown the practical applicability of these policies, since their performance in the real implementation was as good as it was in the simulations. Secondly, we have shown the accuracy of our simulations, since the improvements under the granularity-driven task selection policies obtained for real applications were very similar to the ones for the models of similar applications under the simulations.

Analysis of the usability of load information in state-of-the-art work-stealing algorithms

The state-of-the-art work-stealing algorithms presented in Section 2.3.1 were previously evaluated only for simple divide-and-conquer parallel applications. The established fact was that CRS gives the best speedups for these applications. We have extended the evaluation of these algorithms to highly irregular parallel applications and to highly heterogeneous computing environments. Furthermore, for the first time, we have analysed the scenario in which these algorithms have access to *perfect* information about dynamic loads of the PEs in the computing environment, at each point in the application execution. We have evaluated how much the applications'

speedups under each of these algorithms can be increased when perfect load information is present. This enabled us to estimate how useful it would be to extend each algorithm with a mechanism for estimating PE loads.

Our evaluation showed that the amount of improvement that can be obtained depends on the algorithm used. The performance of some algorithms (e.g. CRS and Adaptive CRS) can be improved with the use of load information only for highly irregular applications on highly heterogeneous computing environments, whereas for others (Hierarchical and Random Stealing) it can be improved for all combinations of applications and computing environments, except for very regular applications on homogeneous computing environments. We have also discovered that the CRS algorithm gives the best speedups for all applications on all computing environments, in the case that it has access to perfect load information.

The benefits of our findings are twofold:

- They give a recommendation for the developers of new work-stealing runtime systems to use the CRS algorithm for irregular parallel applications, assuming that they can incorporate mechanisms for obtaining a good approximation of dynamic PE loads in their systems.

- It gives an estimation of how much speedups can be improved with the use of load information under the individual state-of-the-art work-stealing algorithms we considered in this thesis. This can be useful for the developers of work-stealing runtime systems that use one of the algorithms we considered. The developers can estimate how much improvement in the applications' speedups can be obtained under the algorithm they use, should they decide to use load information.

The precise definition of the degree of irregularity of an application (with respect to task granularity)

We have proposed precise definitions of the *degree of irregularity* of parallel applications, based on the sizes of their tasks. The proposed definitions covered both the applications without nested parallelism (Definition 2) and the applications with nested parallelism (Definition 4). Compared to the usual approach in the literature, where the applications are regarded simply as either regular or irregular, our definitions enable us to compare two irregular applications and decide which one of them is more irregular. We have used these definitions to make a correlation between the effectiveness of work-stealing algorithms and the irregularity of an application.

Highly-parameterizable simulator for work-stealing on heterogeneous computing environments

We have developed the SCALES simulator, which simulates the execution of parallel applications on heterogeneous computing environments under different work-stealing algorithms. This is the first simulator that specifically targets work-stealing on heterogeneous computing environments and it can, therefore, be used to evaluate the effectiveness of work-stealing algorithms for a wide class of applications on a wide class of computing environments. It can serve as an ideal testbed for a new work-stealing algorithm before its implementation is attempted in a realistic runtime system, as this is usually a major task.

7.2 Limitations of our Approach

Most of the limitations of our work come from the specific restrictions we have placed on the types of applications, computing environments and runtime systems that we considered. Specifically, we can point to the following limitations:

- *We considered only the embarrassingly parallel applications*
  A very important restriction on the kind of applications that we considered was that they were embarrassingly parallel. We did not evaluate Feudal Stealing or granularity-driven task selection policies for the applications that have non-trivial data dependencies, but we anticipate that there could be problems during their execution if tasks that need to communicate often are placed on the PEs connected by a high-latency network. Since Feudal Stealing and the granularity-driven task selection policies consider the placement of each task independently of all other tasks, they can run into the aforementioned problem.

- *The only kind of heterogeneity in the computing environment that we considered was the presence of the different communication latencies between the different PEs*
  In distributed computing environments, especially on Computational Grids, heterogeneity can come from different computing capabilities of PEs and a different amount and type of memory associated with them. Additionally, rather than being static, computing capabilities of PEs and communication latencies (and also bandwidths, which we did not consider in this thesis) in the network links can vary over the course of the application execution. We did not address the

heterogeneity in computing capabilities and the variability in the performance of network links in this thesis.

Note that some other kinds of heterogeneity in the computing environment can be modelled as dynamic heterogeneity in communication latencies and computing capability. For example, in Computational Grids and Clouds, computing resources may not be dedicated to one application, and PEs may have a dynamically varying background load. However, changes in the background load can be represented as dynamically varying computing capabilities of PEs, where a capability of a PE decreases when background tasks are added to it. Also, limited bandwidth of a communication link can be modelled as the dynamically changing communication latency of that link, where the link latency increases if too large messages are sent over it.

- *We did not consider runtime systems that allow task migration*
  Besides stealing tasks whose execution has not started, we can also allow thieves to steal tasks whose execution has already started (i.e. we can allow *task migration*). Most of the work-stealing runtime systems do not support task migration, as it can be prohibitively expensive, and this was the reason for not considering task migration mechanism in this thesis. However, the Grid-GUM runtime system allows task migration, and it has been showed that this can dramatically increase the speedups of some applications (see Du Bois et al. [BLT03]).

- *We assumed that only one task is transferred in each steal operation*
  In order to decrease the number of messages exchanged over high-latency networks, we may consider packing more than one task in a message that a victim sends back to a thief as a response to a steal attempt. The evaluation of Grid-GUM (see Al Zain et al. [AZTML08]) showed that this can be beneficial in some situations.

7.3 Further Work

Based on the limitations we described in Section 7.2 and the current trends in computing environments, we can observe several obvious ways in which this thesis can be improved.

Work-stealing algorithms that support applications with more complex task dependencies (e.g. applications with pipeline parallelism)

The first obvious direction for improving the work done in this thesis is to consider the applications that have more data-dependencies between tasks. We can take into account not only the fact that some tasks can share the data, but also provide for different sizes of the input data that tasks need and the output data that they generate (similarly to what is done in the DAG application model). This would require both an extension to the SCALES simulator in order to be able to model this kind of applications, and also an extension to the Feudal Stealing and granularity-driven task selection policies.

In order to extend the SCALES simulator to allow modelling of applications with complex data dependencies, we would need to add a new task event which simulates the communication between two tasks. We can parametrise this event by the amount of communication that needs to be done in terms of, for example, the number of messages that needs to be exchanged or the size of the data that needs to be communicated. Therefore, for example, for a task t we can have the event COMMUNICATE t_1 m, which denotes that task t needs to fetch some data from the task t_1 (or from the PE where t_1 was executed, if t_1 has finished the execution). This fetching would require m messages (or, alternatively, that the data that needs to be fetched is m bytes long). The former approach, where the event is parametrised by the size of the data that needs to be fetch would require us to extend the computing environments we simulate with the concept of a bandwidth between PEs/clusters. In any case, the introduction of COMMUNICATE event would enable us to model applications whose tasks share some data, or where data is continuously passed from one task to the other (e.g. pipeline parallelism).

Concerning the extensions to work-stealing algorithms we proposed, we would need to introduce grouping of tasks based on their data dependencies. We can, for example, decide that a certain set of tasks may be executed only on PEs that communicate over low-latency networks or, in the extreme cases of huge data-dependencies, even on the same PE. Granularity-driven task selection policies can then, in addition to the task sizes, also use this grouping to guide the decision of what task(s) to send to a thief.

Simulating a wider class of computing environments

The second obvious direction for improving the work done here is to consider computational environments where, in addition to communication latencies, heterogeneity also exists in computing capabilities of individual PEs and bandwidths of communi-

cation links. We can also work under the assumption that certain characteristics of computing environments (in terms of, for example, latency and bandwidth of communication links and computing capabilities of individual PEs) dynamically vary over the course of application execution. This would enable us to simulate more accurately the behaviour of non-dedicated Computational Grids. Our work-stealing algorithms would then need to take into the account also these other kinds of heterogeneity. For example, granularity-driven task selection policies may try to avoid sending the tasks that are too large to the PEs whose computing capability is currently low.

Using the task migration in load balancing

In the applications that we considered, it can happen that a target does not have any unstarted tasks in its task pool, but that it has many started tasks. In this situation, rather than considering the steal as unsuccessful, we can decide to send one of the started tasks. This might be the only solution to achieve a load balance for some applications, especially if we consider applications with data dependencies. For example, a situation might happen where a PE has a lot of unstarted tasks that share the same data that is placed on some other PE, and all tasks need this data at the beginning of their execution. In this situation, the PE might start the execution of one task that might immediately block on communication. Since the PE becomes idle again, it can try to execute other tasks, where the same situation can happen. Therefore the PE can accumulate a lot of tasks that are blocked at the beginning of their execution, and they can all be unblocked at approximately the same time (i.e. when they fetch the data they need). This would leave the PE with a lot of started tasks, while other PEs in the computing environment might be idle.

Considering the task migration would require us to carefully weight the overheads in the migration of a task, with respect to the benefits of its parallel execution. Migrating started tasks might be very expensive in terms of the communication time it takes, since it might involve transferring the whole task state from a victim to a thief. We would need, therefore, to provide the estimation of how much communication time would be needed to migrate a task and how much of it is already completed, and probably the overall load of the system in order to be able to decide whether the task migration is worthwhile.

Generally, we might consider task migration as a last resort for achieving load balance, which should be avoided if at all possible. Considering the Feudal work-stealing algorithm, we can consider two separate loads – one that denotes how many unstarted tasks a PE has, and the other that denotes how many started tasks it has.

We can then decide to ask a victim (that has no unstarted tasks) for work only if no other victim (that has started tasks) exists.

Adaptive task selection policies

We saw in Chapter 6 that none of the proposed granularity-driven task selection policies gives the best speedups for all applications that we considered. We can, therefore, consider developing an adaptive task selection policy, which would dynamically switch between different "basic" policies, depending on the number of tasks that remain to be executed and the mean size of the remaining tasks. This adaptive policy would use the SSL policy for most of the application execution, and it would switch to the SLL policy when the number of tasks that remain to be executed becomes low and in the case that the remaining tasks are coarse-grained.

Considering different definitions of PE load

In the discussion about task migration (see "Using the task migration in load balancing"), we pointed out that it might be useful to consider extending the trivial definition of a PE load that was used in Feudal Stealing (where we defined a PE load as the number of unstarted tasks in its task pool) to cover both started and unstarted tasks. This is a part of a more general direction in which work in this thesis can be extended. Namely, we may need to consider non-trivial definitions of a PE load, that would take into account not only the number of tasks that a PE has, but also some characteristics of tasks. For example, incorporating the information about task sizes and the amount of parallelism they generate may allow thieves to select steal targets that will have appropriate work to send. For example, when thieves need to choose remote targets, they may be interested in locating targets that have fewer number of coarse grained tasks that generate a lot of parallelism, rather than those that have a larger number of fine-grained or sequential tasks. On the other hand, when thieves look for work locally (within the same cluster), they might be interested in finding the targets that have finer-grained tasks, as we saw the benefits of this in SSL granularity-driven task selection policy. Finding the appropriate definition of a PE load would then allow Feudal Stealing to find not only *any* target that has work, but also the target that has *the best* work to send.

A combination of shared-memory and distributed-memory work-stealing algorithms

A more major direction for improving the work-stealing algorithms might be combining the shared-memory implementations of work-stealing, where a common task queue for all PEs exists (for example, in the multicore implementation of GpH [MPJS09]) and the algorithms that were used in this thesis. Because of the emergence of multicore and manycore architectures, future large-scale computing environments will probably consist of geographically distributed clusters of multicore/manycore machines. Therefore, we anticipate that the runtime systems will also need to be adapted to this combination of shared- and distributed-memory architectures, and some of them might consider combinations of shared-memory and distributed-memory application models.

A Markov-chain based system for obtaining the profiles of applications' tasks

Granularity driven task selection policies assume the presence of accurate information about the sizes of tasks that comprise an application being executed. Besides the sizes of the tasks, many other task characteristics may be of use in work-stealing/scheduling, e.g. the amount of communication between tasks and the amount of parallelism they generates. Therefore, methods for obtaining this information would be very useful in load-balancing and scheduling.

In our previous work (Janjic et al. [JHY08]), we showed that the *Markov-chain* model can be useful in recognising the patterns of task sizes in bag-of-tasks applications on Computational Grids. We may consider using the similar model in profiling of the application over several runs in order to obtain the estimations of important characteristics of the profiles of its tasks. This can be used as a method for obtaining accurate information about application tasks.

To conclude this thesis, we have developed and evaluated novel work-stealing algorithms specifically tailored to the execution of irregular parallel applications on heterogeneous distributed computing environments, and we have also showed that these algorithms do not bring a decrease in performance for regular parallel applications. we have, therefore, showed that work-stealing is the method of choice for load-balancing in runtime systems for a very large class of applications. We anticipate that the next major step in the development of work-stealing algorithms will be combining shared-memory and distributed-memory algorithms in runtime systems that use the

combination of these two memory models.

Bibliography

[ABB00] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The Data Locality of Work Stealing. In *Proc. of 12th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 1–12. ACM, 2000.

[AHK98] Robert Armstrong, Debra Hengsen, and Taylor Kidd. The Relative Performance of Various Mapping Algorithms is Independent of Sizable Variances in Run-time Predictions. In *Proc. of the 7th IEEE Heterogeneous Computing Workshop*, HCW '98, pages 79–87. IEEE Computer Society, 1998.

[Ama] Amazon EC2. `http://aws.amazon.com/ec2`.

[ATN+00] Kento Aida, Atsuko Takefusa, Hidemoto Nakada, Satoshi Matsouka, Satoshi Sekiguchi, and Umpei Nagashima. Performance Evaluation Model for Scheduling in Global Computing Systems. *The International Journal of High Performance Computing Applications*, 14(3):268–279, 2000.

[AZ06] Abdallah D. Al Zain. *Implementing High-Level Paralelism on Computational Grids.* PhD thesis, Heriot-Watt University, Edinburgh, 2006.

[AZTH+08] Abdallah D. Al Zain, Philip W. Trinder, Kevin Hammond, Alexander Konovalov, Steve Linton, and Jost Berthold. Parallelism without Pain: Orchestrating Computational Algebra Components into a High-Performance Parallel System. In *Proc. of the 2008 International Symposium on Parallel and Distributed Processing with Applications*, ISPA 2008, pages 99–112. IEEE Computer Society, December 2008.

[AZTML06] Abdallah D. Al Zain, Philip W. Trinder, Greg J. Michaelson, and Hans-Wolfgang Loidl. Managing Heterogeneity in a Grid Parallel Haskell. *Scalable Computing: Practice and Experience*, 7(3):9–25, 2006.

[AZTML08] Abdallah D. Al Zain, Philip W. Trinder, Greg J. Michaelson, and Hans-Wolfgang Loidl. Evaluating a High-Level Parallel Language (GpH) for Computational Grids. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):219–233, 2008.

[BBB96] John E. Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proc. of the 7th Workshop on System Support for Worldwide Applications*, pages 165–172. ACM, 1996.

[BC03] Kevin J. Barker and Nikos P. Chrisochoides. An Evaluation of a Framework for the Dynamic Load Balancing of Highly Adaptive and Irregular Parallel Applications. In *Proc. of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 45–59. ACM, 2003.

[BCC+05] Francine Berman, Henry Casanova, Andrew A. Chien, Keith D. Cooper, HHolly Dail, Anshuman Dasgupta, Wei Jaw Deng, Jack J. Dongarra, Lennart S. Johnsson, Ken W. Kennedy, Charles H. Koelbel, B. Liu, X. Liu, Anirbal Mandal, Gabriel Marin, Mark Mazina, John Mellor-Crummey, Ceslo Luis Mendes, A. Olugbile, Jignesh M. Patel, Daniel A. Reed, Zhiao Shi, Otto Sievert, Huaxia Xia, and Asim YarKhan. New Grid Scheduling and Rescheduling Methods in the GrADS Project. *International Journal of Parallel Programming*, 33(2):209–229, June 2005.

[BFH03] Fran Berman, Geoffrey C. Fox, and Anthony J. G. Hey, editors. *Grid Computing : Making the Global Infrastructure a Reality.* John Wiley & Sons, 2003.

[BH04] Fran Berman and Tony Hey. The Scientific Imperative. In Ian Foster and Carl Kesselman, editors, *The Grid : Blueprint for a New Computing Infrastructure, 2nd Edition*, pages 13–24. Morgan Kauffman Publishers, 2004.

[BHC+93] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-parallel Language. In *Proc. of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '93, pages 102–111. ACM, 1993.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '95, pages 207–216. ACM, 1995.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[Ble96] Guy E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[BLMP97] Silvia Breitinger, Rita Loogen, Yolanda Ortega Mallen, and Ricardo Pena. The Eden Coordination Model for Distributed Memory Systems. In *Proc. of the 1997 Workshop on High-Level Programming Models and Supportive Environments*, HIPS '97, pages 120–124. IEEE Computer Society, 1997.

[BLT03] André R. Du Bois, Hans-Wolfgang Loidl, and Philip W. Trinder. Thread Migration in a Parallel Graph Reducer. In *Proc. of the 14th International Conference on Implementation of Functional Languages*, IFL '02, pages 199–214. Springer-Verlag, 2003.

[BM02] Rajkumar Buyya and Manzur Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation : Practice and Experience (CCPE)*, 14(13):1175–1220, 2002.

[BS81] F. Warren Burton and M. Ronan Sleep. Executing Functional Programs on a Virtual Tree of Processors. In *Proc. of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194. ACM, 1981.

[Buy02] Rajkumar Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Computing.* PhD thesis, School of Computer Science and Software Engineering Monash University, Melbourne, Australia, 2002.

[BWC03] Francine Berman, Richard Wolski, and Henri Casanova. Adaptive Computing on the Grid Using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[CJSN03] Junwei Cao, Stephen A. Jarvis, Subhash Saini, and Graham R. Nudd. GridFlow : Workflow Management for Grid Computing. In *Proc. of the 3rd International Symposium on Cluster Computing and the Grid (CCGrid '03)*. IEEE Computer Society, 2003.

[CK88] Thomas L. Casavant and Jon G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[CKK+08] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In *Proc. of the 37th International Conference on Parallel Processing*, ICPP '08, pages 536–545. IEEE Computer Society, 2008.

[CLJ+07] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A Status Report. In *Proc. of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18. ACM, 2007.

[CLQ08] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: A Generic Framework for Large-Scale Distributed Experiments. In *Proc. of the 10th International Conference on Computer Modeling and Simulation*, ICCMS '08, pages 126–131. IEEE Computer Society, 2008.

[CLZB00] Henry Casanova, Arnaud Lergand, Dmitrii Zagorodnov, and Francine Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proc. of the 9th Heterogeneous Computing Workshop*, HCW '00, pages 349–363. IEEE Computer Society, 2000.

[COBW00] Henri Casanova, Graziano Obertelli, Francine Berman, and Richard Wolski. The AppLeS Parameter Sweep Template: User-level Middleware for the Grid. *Journal of Scientific Programming*, 8(3):111–126, August 2000.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, 1989.

[Col99] Murray Cole. Algorithmic Skeletons. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*. Springer, 1999.

[CRB+11] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, Cesar A. F. De Rose, and Rajkumar Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Software : Practice and Experience*, 41(1):23–50, January 2011.

[DA98] Sekhar Darbha and Dharma P. Agrawal. Optimal Scheduling Algorithm for Distributed-Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):87–95, January 1998.

[DA06] Fangpeng Dong and Selim G. Akl. Scheduling Algorithms for Grid Computing: State of the Art and Open Problems. Technical Report 2006-504, Queen's University, Kingston, Ohio, 2006.

[DBG+03] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Metha, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, and Scott Koranda. Mapping Abstract Complex Workflows onto Grid Environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[DLS+09] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable Work Stealing. In *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 1–11. ACM, 2009.

[ELZ86] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, 1986.

[FAF] FAFNER Project. `http://www.npac.syr.edu/factoring.html`.

[FGN+03] Ian Foster, Jonathan Geisler, Bill Nickless, Warren Smith, and Steven Tuecke. Software Infrastructure for the I-WAY High-Performance Distributed Computing Experiment. In *Grid Computing: Making the Global Infrastructure a Reality*, pages 101–115. John Wiley & Sons, 2003.

[FH88] Anthony J. Field and Peter G. Harisson. *Functional Programming.* Addison-Wesley Publishing Company, 1988.

[FK04a] Ian Foster and Carl Kesselman. Concepts and Architecture. In Ian Foster and Carl Kesselman, editors, *The Grid : Blueprint for a New Computing Infrastructure, 2nd Edition*, pages 37–64. Morgan Kauffman Publishers, 2004.

[FK04b] Ian Foster and Carl Kesselman, editors. *The Grid : Blueprint for a New Computing Infrastructure, 2nd Edition.* Morgan Kauffman Publishers, 2004.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid - Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(4):200–222, 2001.

[FKT04] Ian Foster, Carl Kesselman, and Steven Tuecke. The Open Grid Services Architecture. In Ian Foster and Carl Kesselman, editors, *The Grid : Blueprint for a New Computing Infrastructure, 2nd Edition*, pages 215–249. Morgan Kauffman Publishers, 2004.

[Fos03] Ian Foster. *The Grid : A New Infrastructure for 21st Century Science*, chapter 2. Wiley, 2003.

[FZRL08] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Proc. of the IEEE Grid Computing Environments Workshop*, GCE '08, pages 1–10. IEEE Computer Society, 2008.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1999.

[GLS03] William Gropp, Ewing Lusk, and Thomas Sterling. *Beowulf Cluster Computing with Linux, 2nd Edition.* MIT Press, 2003.

[Goo] Google App Engine. `http://code.google.com/appengine`.

[GS06] Clemens Grelck and Sven-Bodo Scholz. SAC: A Functional Array Language for Efficient Multi-threaded Execution. *International Journal of Parallel Programming*, 34(4):383–427, August 2006.

[GVC06] Horacio Gonzaléz-Vélez and Murray Cole. Towards Fully Adaptive Pipeline Parallelism for Heterogeneous Dsitributed Environments. In *Proc. of the 2006 International Symposium on Parallel and Distributed Processing and Applications*, ISPA 2006, pages 916–926. Springer-Verlag LNCS 4330, 2006.

[GVC10] Horacio González-Vélez and Murray Cole. Adaptive Statistical Scheduling of Divisible Workloads in Heterogeneous Systems. *Journal of Scheduling*, 13(4):427–441, 2010.

[GWT97] Andrew S. Grimshaw, William. A. Wulf, and The Legion Team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[Hag97] Torben Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *Journal of Parallel and Distributed Computing*, 47(2):185–197, 1997.

[HB95] Jeffrey Hammes and Wim Bohm. Comparing Id and Haskell in a Monte Carlo Photon Transport Code. *Journal of Functional Programming*, 5:283–316, 1995.

[HC07a] Israel Hernandez and Murray Cole. Reliable DAG Scheduling on Grids with Rewinding and Migration. In *Proc. of the 1st International Conference on Networks for Grid Applications*, GridNets '07, pages 3:1–3:8. ICST, 2007.

[HC07b] Israel Hernandez and Murray Cole. Scheduling DAGs on Grids with Copying and Migration. In *Proc. of the 7th International Conference on Parallel Processing and Applied Mathematics*, PPAM '07, pages 1019–1028. Springer-Verlag LNCS 4967, 2007.

[Her01] Christoph A. Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions.* PhD thesis, University of Passau, Germany, 2001.

[HM99] Kevin Hammond and Greg Micaelson, editors. *Research Directions in Parallel Functional Programming.* Springer-Verlag, 1999.

[HSF92] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*, 35(8):90–101, 1992.

[Hug90] John Hughes. *Research Topics in Functional Programming*, chapter Why Functional Programming Matters, pages 17–42. Addison-Wesley Longman Publishing Co., Inc., 1990.

[HYUY09] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based Load Balancing. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 55–64. ACM, 2009.

[Jay99] C. Barry Jay. Shaping Distributions. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 219–232. Springer, 1999.

[JH10] Vladimir Janjic and Kevin Hammond. Granularity-Aware Work-Stealing for Computationally-Uniform Grids. In *Proc. of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGrid '10, pages 123–134. IEEE Computer Society, May 2010.

[JHH+92] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glagow Haskell Compiler : A Technical Overview. In *Proc. of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1992.

[JHY08] Vladimir Janjic, Kevin Hammond, and Yang Yang. Using Application Information to Drive Adaptive Grid Middleware Scheduling Decisions. In *Proc. of the 2nd Workshop on Middleware-application Interaction: Affiliated with the DisCoTec Federated Conferences 2008*, MAI '08, pages 7–12. ACM, 2008.

[Jon92] Simon L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2:127–202, 1992.

[Jon03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, December 2003.

[JWT+04] Seung-Hye Jang, Xingfu Wu, Valerie Taylor, Gaurang Metha, Karan Vahi, and Ewa Deelman. Using Performance Prediction to Allocate Grid Resources. Technical report, Texas A&M University and UCS Information Sciences Institute, 2004.

[KBAK10] Dzmitry Kliazovich, Pascal Bouvry, Yury Audzevich, and Samee Ulah Khan. GreenCloud: A Packet-level Simulator of Energy-aware Cloud Computing Data Centers. In *Proc. of the 53rd IEEE Global Communication Conference*, GLOBECOM 2010, pages 1–5. IEEE Computer Society, 2010.

[KGV94] Vipin Kumar, Ananth Y. Grama, and Nageshwara Rao Vempaty. Scalable Load Balancing Techniques for Parallel Computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

[KPW+07] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic Parallelism Requires Abstractions. In *Proc. of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222. ACM, 2007.

[KT99] Paul Kelly and Frank Taylor. Coordination Languages. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 305–320. Springer, 1999.

[KTF03] Nicholas T. Karonis, Brian Toonen, and Ian Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, May 2003.

[KW85] Clyde P. Kruskal and Alan Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, 1985.

[LH97] Hans-Wolfgang Loidl and Kevin Hammond. Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer. In *Proc. of the 8th International Workshop on Implementation of Functional Languages*, IFL '96, pages 184–199. Springer-Verlag, 1997.

[LHK+11] Stephen Linton, Kevin Hammond, Alexander Konovalov, Philip W. Trinder, Christopher Brown, Hans-Wolfgang Loidl, Peter Horn, and Dan

Roozemond. Easy Composition of Symbolic Computation Software using SCSCP: A New Lingua Franca for Cymbolic Computation. *To Appear in Journal of Symbolic Computation*, 2011.

[Loi98] Hans-Wolfgang Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, University of Glasgow, 1998.

[Loi02] Hans-Wolfgang Loidl. Load Balancing in a Parallel Graph Reducer. In *Proc. of the International Symposium on Trends in Functional Programming*, TFP '02, pages 63–74. Intellect Books, 2002.

[Loo99] Rita Loogen. Programming Language Constructs. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 63–91. Springer, 1999.

[MAS+99] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic Matching and Scheduling of a Class of Indipendent Tasks onto Heterogeneous Computing Systems. In *Proc. of the 8th IEEE Heterogeneous Computing Workshop*, HCW '99, pages 30–44. IEEE Computer Society, 1999.

[MCGH05] Anne Benoit Murray, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible Skeletal Programming with eSkel. In *Proc. of the 11th Euro-Par: Parallel and Distributed Computing*, Euro-Par 2005, pages 761–770. Springer-Verlag LNCS 3648, 2005.

[MLD06] Stephen McGough, William Lee, and John Darlington. Workflow Deployment in ICENI II. In *Proc. of the 6th International Conference on Computational Science*, ICCS 2006, pages 964–971. Springer-Verlag LNCS 3993, 2006.

[MLS+05] Nithiapidary Muthuvelu, Junyang Liu, Nay Lin Soe, Srikumar Venugopal, Anthony Sulistio, and Rajkumar Buyya. A Dynamic Job Grouping-based Scheduling for Deploying Applications with Fine-grained Tasks on Global Grids. In *Proc. of the 2005 Australasian Workshop on Grid Computing and E-research*, volume 44 of *ACSW Frontiers '05*, pages 41–48. Australian Computer Society Inc., 2005.

[MPJS09] Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. Runtime Support for Parallel Haskell. In *Proc. of the 14th ACM SIGPLAN In-*

ternational Conference on Functional Programming, ICFP 2009, pages 65–78. ACM, 2009.

[MSA] Microsoft Windows Azure. `http://www.microsoft.com/windowsazure`.

[MVS09] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswar. Idempotent Work Stealing. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 45–54. ACM, 2009.

[NA01] Rishiyur S. Nikhil and Arvind. *Implicit Parallel Programming in pH.* Morgan Kaufmann Publishers Inc., 2001.

[NC02] Michael O. Neary and Peter Cappello. Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. In *Proc. of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, JGI '02, pages 56–65. ACM, 2002.

[NIS] NIST Definition of Cloud Computing v15. `csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc`.

[NKP+00] Graham R. Nudd, Darren J. Kerbyson, Efstathios Papaefstathiou, S. C. Perry, John S. Harper, and Daniel V. Wilcox. Pace–A Toolset for the Performance Prediction of Parallel and Distributed Systems. *International Journal of High Performance Computing Applications*, 14(3):228–251, 2000.

[NPA01] Dimitrios S. Nikolopoulos, Constantine D. Polychronopoulos, and Eduard Ayguadé. Scaling Irregular Parallel Codes with Minimal Programming Effort. In *Proc. of the 2001 ACM/IEEE Conference on Supercomputing*, Supercomputing '01, pages 16–16. ACM, 2001.

[PBS] Open portable batch system. `http://www.mcs.anl.gov/research/projects/openpbs`,.

[PCM+07] Guilherme P. Pezzi, Marcia C. Cera, Elton Mathias, Nicolas Maillard, and Philippe O. A. Navaux. On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing. SBAC-PAD 2007, pages 247–254. IEEE Computer Society, 2007.

[PvEPS99] Rinus Plasmeijer, Marko van Eekelen, Marco Pil, and Pascal Serrarens. Parallel and Distributed Programming in Concurrent Clean. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 323–338. Springer, 1999.

[Qui03] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[RBJ03] David De Roure, Mark A. Baker, and Nicholas R. Jennings. The Evolution of the Grid. In *Grid Computing: Making the Global Infrastructure a Reality*, pages 65–100. John Wiley & Sons, 2003.

[RCL09] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *Proc. of the 5th International Joint Conference on INC, IMS and IDC*, NCM '09, pages 44–51. IEEE Computer Society, 2009.

[RvG99] Andrei Radulescu and Arjan J.C. van Gemund. On the Complexity of List Scheduling Algorithms for Distributed-Memory Systems. In *Proc. of the 13th International Conference on Supercomputing*, ICS '99, pages 68–75. ACM, 1999.

[Ś99] Jocelyn Sérot. Explicit Parallelism. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 379–396. Springer, 1999.

[Sal] Salesforce.com. `http://www.salesforce.com`.

[Sar04] Vijay Saraswat. X10 Language Report. Technical report, IBM, 2004.

[Sha01] Gary Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources.* PhD thesis, University of California, San Diego, 2001.

[Ske91] Stephen K. Skedzielewski. Sisal. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–157. ACM, 1991.

[SKS92] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load Distributing for Locally Distributed Systems. *Computer*, 25(12):33–44, 1992.

[SLJ+00] Hyou J. Song, Xin Liu, D. Jakobsen, R. Bhagwan, K. Zhang, Kenjiro Taura, and Andrew A. Chien. The MicroGrid: A Scientific Tool for Modeling Computational Grids. *Journal of Scientific Programming*, 8(3):127–141, August 2000.

[SZ04] Rizos Sakellariou and Henan Zhao. A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems. In *Proc. of the 18th International Parallel and Distributed Processing Symposium*, IPDPS '04, pages 111–123. IEEE Computer Society, 2004.

[THLPJ98] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

[THM+96] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: A Portable Parallel Implementation of Haskell. In *Proc. of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '96, pages 79–88. ACM, 1996.

[THW02] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[TL04] Douglas Thain and Miron Livny. Building Reliable Clients and Services. In Ian Foster and Carl Kesselman, editors, *The Grid : Blueprint for a New Computing Infrastructure, 2nd Edition*, pages 285–318. Morgan Kauffman Publishers, 2004.

[TWS03] Valerie Taylor, Xingfu Wu, and Rick Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *ACM SIGMETRICS Performance Evaluation Review*, 30:13–18, March 2003.

[VMW] VMWare Cloud Infrastructure & Management Solutions. `http://www.vmware.com/solutions/cloud-solutions.html`.

[VNKB01] Rob V. Van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient Load Balancing for Wide-area Divide-and-Conquer Applications. In

Proc. of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, PPoPP '01, pages 34–43. ACM, 2001.

[VNMW+04] Rob V. Van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Thilo Kielmann, and Henri E. Bal. Adaptive Load Balancing for Divide-and-Conquer Grid Applications. *Journal of Supercomputing*, 2004.

[VNWJB10] Rob V. Van Nieuwpoort, Gosia Wrzesińska, Ceriel J. H. Jacobs, and Henri E. Bal. Satin: A High-Level and Efficient Grid Programming Model. *ACM Transactions on Programming Languages and Systems*, 32(3):1–39, 2010.

[VRMCL09] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, January 2009.

[Xen] XenSource Inc. `www.xensource.com`.

[YB05] Jia Yu and Rajkumar Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *ACM SIGMOD Record*, 34(3):44–49, September 2005.

[YBDS08] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a Unified Ontology of Cloud Computing. In *Proc. of the 2008 Workshop on Grid Computing Environments*, GCE '08, pages 1–10. IEEE Computer Society, November 2008.

[YG94] Tao Yang and Apostolos Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: State-of-the-Art and Research Challenges. *Journal of Internet Services and Applications*, 1(1):7–18, May 2010.

[ZKK07] Yang Zhang, Charles Koelbel, and Ken Kennedy. Relative Performance of Scheduling Algorithms in Grid Environment. In *Proc. of the 7th IEEE International Symposium on Cluster Computing and the Grid*, CCGrid '07, pages 521–528. IEEE Computer Society, May 2007.