Abstract of “ Theory and Applications of Parallelism with Futures
” by Zhiyu Liu , Ph.D., Brown University, May 2017.

Futures are an attractive way to structure parallel computations. When a thread creates an expression with a keyword future, a new thread is spawned to compute that expression in parallel. When a thread later applies a touch operation to that future, it gets the result of the expression if the result has been computed, and otherwise blocks until the result becomes ready. In this thesis, we explore different aspects of parallel programs with futures, including their theoretical bounds, scheduling, and applications.

Researchers have shown futures can have a deleterious effect on cache locality. We will show, however, that if futures are used in a simple, disciplined way, then their negative impact can be much alleviated. This structured use of futures is characteristic of many parallel applications.

Futures lend themselves well to dynamic scheduling algorithms, such as work stealing, that ensure high processor utilization. Implementing work stealing on hierarchical platforms, such as NUMA systems and distributed clusters, have recently drawn lots of attention. However, little has been explored on its theoretical bounds. We present lower and upper bounds of work stealing for fork-join programs, a well-studied subclass of parallel-future programs, on hierarchical systems.

As originally conceived, a future encapsulates a functional computation without side-effects. Recently, however, futures have been proposed as a way to encapsulate method calls to shared data structures. We propose a new program model, that supports both normal futures without side-effects and linearizable futures that exist for their side-effects. Using this model, we propose the lazy work stealing scheduler that facilitates certain optimizations for linearizable futures and guarantees good time bounds.

The processing-in-memory (PIM) model has reemerged recently as a solution to alleviating the growing speed discrepancy between CPU’s computation and memory access, commonly known as the memory wall. In this model, some lightweight computing units are directly attached to the main memory, providing fast memory access. We study applications of linearizable futures in the PIM model: operation requests to concurrent data structures are sent as linearizable futures to those computing units to execute. These PIM-managed data structures can outperform state-of-the-art concurrent data structures in the literature.

Theory and Applications of Parallelism with Futures

by
Zhiyu Liu
B. E., Beijing University of Posts and Telecommunications, 2010
Sc. M., Dartmouth College, 2012

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2017

This dissertation by Zhiyu Liu is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ______________ ______________________________
Maurice Herlihy, Director

Recommended to the Graduate Council

Date ______________ ______________________________
Rodrigo Fonseca, Reader
Brown University

Date ______________ ______________________________
Eli Upfal, Reader
Brown University

Approved by the Graduate Council

Date ______________ ______________________________
Andrew G. Campbell
Dean of the Graduate School

Vita

Education

- **Ph.D** in Computer Science, Brown University, September 2012 – 2017

  Advisor: Maurice Herlihy

- **M.S.** in Computer Science, Dartmouth College, September 2010 – June 2012

  Advisor: Prasad Jayanti

  Thesis: Abortable Reader-Writer Locks Are No More Complex Than Abortable Mutex Locks

- **B.Eng.** in Computer Science and Technology, Beijing University of Posts and Telecommunications, September 2006 – July 2010

Work Experience

- **Research Intern** in System Algorithms Research Group at Microsoft Research Asia, May 2015 – August 2015

  Mentor: Thomas Moscibroda

  Project: Automatic and efficient scheduling for dependent cloud tasks on Microsoft Azure

- **Research Intern** at VMware Research Group, June 2016 – August 2016

  Mentor: Irina Calciu

  Project: Concurrent data structures in the processing-in-memory model

Publications

- **PIM-managed Concurrent Data Structures**

  Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu

  ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) 2017

- **Well-Structured Futures and Cache Locality**

  Maurice Herlihy and Zhiyu Liu

  ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) 2014

  (Best Paper Award)

- **Approximate Local Sums and Their Applications in Radio Networks**

  Zhiyu Liu and Maurice Herlihy

  International Symposium on DIStributed Computing (DISC) 2014

- **Abortable Reader-Writer Locks Are No More Complex Than Abortable Mutex Locks**

  Prasad Jayanti and Zhiyu Liu

  International Symposium on DIStributed Computing (DISC) 2012

Manuscripts Related to Thesis

- **Theoretical Analysis of Work Stealing on Hierarchical Platforms**

  Zhiyu Liu and Maurice Herlihy

- **Work Stealing for Linearizable Futures**

  Zhiyu Liu and Maurice Herlihy

Acknowledgements

First, I would like to thank my advisor, Maurice Herlihy, who has been a perfect mentor throughout the five years of my PhD life. He gave me everything I needed in research: guidance, support, inspiration, vision, and freedom. I was incredibly lucky to be supervised by such a world-class researcher.

I would also like to thank my thesis committee members, Rodrigo Fonseca and Eli Upfal, for their support and all the nice discussions. I wish to thank Irina Calciu and Thomas Moscibroda, my mentors during my internships at VMware and Microsoft Research, for the wonderful experiences in the two summers.

I want to thank all my colleague students I have worked with: Archita Agarwal, Esha Ghosh, Eli Rossenthal, Vikram Saraph, and Hammurabi Mendes. I wish we could have even more collaborations in the future.

I would thank all other faculty members and students in our department. One of the most enjoyable things to me at Brown was to attend their talks and lectures. It is a great honor for me to know and learn from them.

I also want to thank all the administrative and technical staff in our department for all their support over the five years. In particular, my special thanks go to Lauren Clarke and Eugenia DeGouveia who helped me countless times during the process of scheduling my proposal and defense.

Finally, I thank all my family and friends, in the U.S. and back in China, for everything they have done for me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and Motivations

Futures [41, 42] are an attractive way to structure many parallel programs. When a thread creates an expression with a keyword *future*, a new thread is spawned to compute that expression in parallel with the thread that created it. When a thread later applies a *touch* operation to that future, it gets the result of the expression if the result has been computed, and otherwise blocks until the result becomes ready. Futures were first proposed by Halstead [41, 42] and have been well studied since then (e.g., [55, 8, 58, 31, 16, 7, 20, 77, 32, 1, 73]), sometimes under different names.

Futures lend themselves well to sophisticated dynamic scheduling algorithms, such as work stealing [20] and its variations, that ensure high processor utilization and hence high execution speedup. Arora et al. [7] proved that (parsimonious) work stealing achieves the asymptotically optimal speedup for the parallel execution of a program in the future-parallel model.

As originally conceived, a future encapsulates a short-lived functional computation that has no side-effects, so the order in which futures are execute cannot be observed. Recently, Kogan and Herlihy [53] proposed an alternative approach, in which futures encapsulate method calls to long-lived shared data structures, facilitating common optimizations such as combining [37, 39, 46] and elimination [45, 63, 72]. Since operations on shared objects are typically executed for their side-effects, they also proposed several variations of linearizability [49] to constrain how the computations of futures with side-effects can be interleaved.

Despite a rich literature of research on future-parallel programs, we find the following interesting problems unexplored.

- The asymptotically optimal execution time of programs with futures [7] is proved without considering cache performance. However, modern multicore architectures employ complex multi-level memory hierarchies, and technology trends are increasing the relative performance differences among the various levels of memory. As a result, processor utilization can no longer be the sole figure of merit for schedulers and the *cache locality* of a parallel execution will become increasingly critical to overall performance.

Several researchers [1, 73] have shown, however, that introducing parallelism through the use of futures can sometimes substantially reduce cache locality. In the worst case, if we add futures to a sequential program, a parallel execution managed by a work stealing scheduler can incur $\Omega(PT_\infty + tT_\infty)$ deviations, which implies $\Omega(CPT_\infty + CtT_\infty)$ more cache misses than the sequential execution. Here, C is the number of cache lines, P is the number of processors, t is the number of touch operations, and T_∞ is the computation's *span* (or *critical path*). As technology trends cause the cost of cache misses to increase, this additional cost is troubling. Therefore, we believe it will be very interesting to study new ways of using futures, such as constructing parallel programs in structured forms or designing cache-friendly schedulers for parallel executions, to improve the cache performance of programs with futures.

- As hierarchical platforms, such as NUMA systems and distributed clusters, are becoming more and more prevalent, implementing efficient work stealing algorithms for parallel programs on such hierarchical systems have drawn lots of attention recently. Different techniques have been proposed to improvement the performances of different work stealing algorithms on hierarchical systems. For example, researchers have found different heuristic strategies on when a processor should make a remote steal and how much work it should steal (e.g., [62, 70, 65, 76]). Other people have presented work stealing variants via message passing [29, 71, 78, 2, 57], as opposed to the traditional ways based on concurrent data structures.

  Despite the rich research on the practical implementations and empirical analysis of work stealing on hierarchical systems, however, little has been explored with respect to their theoretical bounds. Quintin and Wagner [70] gave a theoretical bound on the time complexity of their hierarchical work stealing (HWS) algorithm, in which the leader processors of clusters first distribute tasks across a system and then the worker processors in each cluster execute their local tasks. However, without good knowledge of a program, the workloads distributed to different clusters can be highly unbalanced in theory, and hence the upper bound of HWS cannot help us figure out the performance bounds of work stealing.

- As we mentioned earlier, Kogan and Herlihy [53] proposed a new type of futures, which we call *linearizable futures*, to encapsulate method calls to long-lived shared data structures. Their original proposal left open the problem of how best to schedule programs that employ linearizable futures with side-effects.

  The idea behind the linearizable futures is to delay the executions of data structures' method calls, so that certain threads can batch them and apply different optimizations to them, in order to improve overall throughput. Therefore, another interesting problem is to design new data structures on certain platforms that can utilize linearizable futures well to achieve good performance.

1.2 Overview of Contributions

This thesis makes fours contributions, as summarized in Sections 1.2.1-1.2.4 blow.

1.2.1 Well-Structured Futures and Cache Locality

In Chapter 2, we will study the cache performance of future-parallel programs and show that if futures are used in a simple, disciplined way, then the situation with respect to cache locality is much better: if each future is touched only once, either by the thread that created it, or by a thread to which the future has been passed directly or indirectly from the thread that created it, then parallel executions with work stealing can incur at most $O(CPT_\infty^2)$ additional cache misses, a substantial improvement over the unstructured case. This result provides a simple way to identify computations for which introducing futures will not incur a high cost in cache locality, as well as providing guidelines for the design of future parallel computations. (Informally, we think these guidelines are natural, and correspond to structures programmers are likely to use anyway.) We also prove that this upper bound is tight within a factor of C.

Our second result is the observation that when a work stealing scheduler has a choice between running the thread that created a future, and the thread that implements the future, running the future thread first provides better cache locality.

Finally, we show that certain variations of structured computation also have good cache locality.

1.2.2 Theoretical Analysis of Work Stealing on Hierarchical Platforms

In Chapter 3, we will present lower bounds for all work stealing algorithms as well as upper bounds for specific work stealing variants in a theoretical hierarchical model. More specifically, we consider a hierarchical system model of k homogeneous clusters, each having n local processors. The execution of an instruction of a program and an operation for local communication both take time 1, while a remote operation, such as a remote steal and a remote join of two threads, takes time $\Theta(r)$. We focus on fork-join programs [15], a well-studied subclass of future-parallel programs. Our hierarchical system model is kept abstract and general, in order to cover both hierarchical shared-memory and message-passing systems.

We prove the lower bound on the execution time of any load balancing algorithm in this model is $\Omega(\min\{\frac{T_1}{n} + T_\infty, \frac{T_1}{p} + \frac{rT_\infty}{\log(nr)}\})$, where T_1 and T_∞ are the total work and the critical path length of a fork-join program, respectively, and $p = kn$ is the number of processors in the whole system. This lower bound indicates that, when $\frac{T_1}{T_\infty} \leq \frac{nr}{\log(nr)}$, i.e., when the workload of a program is light, running a single cluster of n processors with the classical work stealing algorithm can achieve an optimal expected execution time $O(\frac{T_1}{n} + T_\infty)$.

when $\frac{T_1}{T_\infty} > \frac{nr}{\log(nr)}$, i.e., when the workload is heavy, we show that an algorithm, called global work stealing algorithm, can achieve an execution time $O(\frac{T_1}{p} + rT_\infty)$ in expectation, which is optimal within a factor of $O(\log(nr))$. This algorithm is essentially the classical work stealing algorithm with the "attaching scheme", which resembles the clone optimization [34], to reduce remote joins. Its upper bound is a little surprising, as processors in the global work stealing algorithm choose victim processors uniformly at random, which is not considered a good strategy on hierarchical systems.

In most fork-join programs we find in practice, a thread stops splitting into two parallel ones when the amount of work is small enough and can be quickly done sequentially. Thus to analyze work stealing on these programs, we also define two subclasses of fork-join programs, called s-bounded fork-join programs and s-bounded divide-and-conquer programs, where each thread has at least s nodes of work. We show that a very similar lower bound $\Omega(\min\{\frac{T_1}{n} + T_\infty, \frac{T_1}{p} + \frac{rT_\infty}{\log(\frac{nr}{s})}\})$ still holds for all work stealing algorithms, for any $s = O(r)$. On the other hand, we prove that the class of unbalanced work stealing algorithms can achieve a good upper bound $O(\frac{T_1}{p} + rT_\infty)$, any for $s = \Omega(r)$. This bound is optimal within a factor of $\log(n)$, when the workload is heavy, i.e., $\frac{T_1}{T_\infty} > \frac{nr}{\log(n)}$. We believe the class of unbalanced work stealing algorithms capture the characteristics of many work stealing algorithms using the heuristic that processors should make more local steals than remote steals, a commonly used strategy found empirically effective in practice [62]. Therefore, our result may imply that this heuristic, combined with the strategy of having threads stop splitting early, is likely to have good performance guarantees.

1.2.3 Work Stealing for Linearizable Futures

In chapter 4, we will propose a new program model, called the *linearizable-futures model*, that supports both futures without side-effects, which we call *normal futures*, and futures that exist for their side-effects, which we call *linearizable futures*. This model requires futures to be used in certain structured ways. These constraints are reasonable, in the sense that they rule out only certain pathological uses that seem unlikely to occur in practice.

We use this model to propose a novel scheduler, called *lazy work stealing*, a variant of the classical work stealing intended to facilitate combining and elimination optimizations for linearizable futures.

Finally, we prove bounds on program execution time using lazy work stealing. We show that if the execution time of a program by an optimal offline scheduler is $\Theta(\frac{T_1}{P_A} + T_\infty)$, the expected execution time by lazy work stealing is $O(\frac{T_1}{P'_A} + (c+1)T_\infty)$, where c is the *containment level* of the program, defined below. Roughly speaking, when the containment level c is a small constant, which is the case for many programs, the performance of lazy work stealing is close to that of an optimal offline scheduler. We also show that this bound is asymptotically optimal for non-clairvoyant schedulers, by proving a matching lower bound.

1.2.4 Concurrent Data Structures for Near-Memory Computing

The performance gap between memory and CPU has grown exponentially. Memory vendors have focused mostly on improving memory capacity and bandwidth, sometimes even at the cost of increased memory access latencies. To provide higher bandwidth with lower access latencies, hardware architects have proposed near-memory computing (also called *processing-in-memory*, or PIM), where a lightweight processor (called a PIM core) is located close to memory. A memory access from a PIM core is much faster than from a CPU core. Near-memory computing is an old idea, that has been intensely studied in the past (e.g., [74, 54, 36, 68, 67, 52, 40]), but so far has not yet materialized. However, new advances in 3D integration and in die stacked memory make near-memory computing viable in the near future. For example, one PIM design assumes memory is organized in multiple vaults, each having an in-order PIM core to manage it. These PIM cores can communicate through message passing, but do not share memory, and cannot access each other's vaults.

This new technology promises to revolutionize the interaction between computation and data, as memory becomes an active component in managing the data. Therefore, it invites a fundamental rethinking of basic data structures and promotes a tighter dependency between algorithmic design and hardware characteristics.

Prior work has already shown significant performance improvements by using PIM for embarrassingly parallel and data-intensive applications [79, 4, 80, 6], as well as for pointer-chasing traversals [50] in *sequential* data structures. However, current server machines have hundreds of cores; algorithms for concurrent data structures exploit these cores to achieve high throughput and scalability, with significant benefits over sequential data structures (e.g., [33, 69, 75, 48]).

As we will show, naive PIM data structures cannot outperform state-of-the-art *concurrent* data structures. In particular, the lower latency access to memory cannot compensate for the loss of parallelism. To be competitive with traditional concurrent data structures, PIM data structures need new algorithms and new approaches to leverage parallelism. In Chapter 5, we will present some PIM-managed concurrent data structures, where threads send their operation requests as linearizable futures to PIM cores which execute those requests with certain optimizations.

In particular, we analyze pointer chasing data structures, which have a high degree of inherent parallelism and low contention, but incur significant overhead due to unpredictable memory accesses. We propose using techniques such as combining and partitioning the data across vaults to reintroduce parallelism for these data structures.

Second, we explore contended data structures, such as FIFO queues, which can leverage CPU caches to exploit their inherent high locality. Therefore, FIFO queues might not seem to be able to leverage PIM's faster memory accesses. Nevertheless, these data structures exhibit a high degree of contention, which makes it difficult even for the most advanced algorithms to obtain good performance for many threads accessing the data oncurrently. We use pipelining of requests, which can be done very efficiently in PIM, to design a new FIFO queue suitable for PIM that can outperform state-of-the-art concurrent FIFO queues [64, 44].

Chapter 2

Well-Structured Futures and Cache Locality

In this chapter, we will show that if futures are used in a simple, disciplined way, then their negative impact on cache locality can be much alleviated, a significant improvement over the previous result. This chapter is organized as follows. Section 2.1 describes the model for future-parallel computations. In Section 2.2, we describe parsimonious work-stealing schedulers, and briefly discuss their cache performance measures. In Section 2.3, we define some restricted forms of structured future-parallel computations. Among them, we highlight structured single-touch computations, which, we believe, are likely to arise naturally in many programs. In Section 2.4.1, we prove that work-stealing schedulers on structured single-touch computations incur only $O(CPT_\infty^2)$ additional cache misses, if a processor always chooses the future to execute first when it creates that future. We also prove this bound is tight within a factor of C. In Section 2.4.2, we show that if a processor chooses the current thread over the future thread when it creates that future, then the cache locality of a structured single-touch computation can be much worse. In Section 2.5, we show that some other kinds of structured future-parallel computations also achieve relatively good cache locality.

2.1 Model

In *fork-join parallelism* [15, 13, 17], a sequential program is split into a directed acyclic graph of *tasks* linked by directed dependency edges. These tasks are executed in an order consistent with their dependencies, and tasks unrelated by dependencies can be executed in parallel. Fork-join parallelism is well-suited to dynamic load-balancing techniques such as *work stealing* [22, 41, 7, 20, 1, 42, 18, 34, 55, 3, 23].

A popular and effective way to extend fork-join parallelism is to allow threads to create *futures* [41, 42, 8, 16, 32]. A future is a data object that represents a *promise* to deliver the result of an asynchronous computation when it is ready. That result becomes available to a thread when the thread *touches* that future, blocking if necessary until the result is ready. Futures are attractive because they provide greater flexibility than fork-join programs, and they can also be implemented effectively using dynamic load-balancing techniques such as work stealing. Fork-join parallelism can be viewed as a special case of future-parallelism, where the `spawn` operation is an implicit future creation, and the `sync` operation is an implicit touch of the untouched futures created by a thread. Future-parallelism is more flexible than fork-join parallelism, because the programmer has finer-grained control over touches (joins).

2.1.1 Computation DAG

A thread creates a future by marking an expression (usually a method call) as a *future*. This statement spawns a new thread to evaluate that expression in parallel with the thread that created the future. When a thread needs access to the results of the computation, it applies a *touch* operation to the future. If the result is ready, it is returned by the touch, and otherwise the touching thread blocks until the result becomes ready. Without loss of generality, we will consider fork-join parallelism to be a special case of future-parallelism, where forking a thread creates a future, and joining one thread to another is a touch operation.

Our notation and terminology follow earlier work [7, 20, 1, 73]. A future-parallel computation is modeled as a *directed acyclic graph* (DAG). Each node in the DAG represents a task (one or more instructions), and an edge from node u to node v represents the dependency constraint that u must be executed before v. We follow the convention that each node in the DAG has in-degree and out-degree either 1 or 2, except for a distinguished *root node* with in-degree 0, where the computation starts, and a distinguished *final node* with out-degree 0, where the computation ends.

There are three types of edges:

- *continuation edges*, which point from one node to the next in the same thread,
- *future edges* (sometimes called *spawn* edges), which point from node u to the first node of another thread spawned at u by a future creation,
- *touch edges* (sometimes called *join* edges), directed from a node u in one thread t to a node v in another thread, indicating that v touches the future computed by t.

A *thread* is a maximal chain of nodes connected by continuation edges. There is a distinguished *main thread* that begins at the root node and ends at the final node, and every other thread t begins at a node with an incoming future edge from a node of the thread that spawns t. The last node of t has only one outgoing edge which is a touch edge directed to another thread, while other nodes of t may or may not have incoming and outgoing touch edges. A *critical path* of a DAG is a longest directed path in the DAG, and the DAG's *computation span* is the length of a critical path.

As illustrated in Figure 2.1, if a thread t_1 spawns a new thread t_2 at node v in t_1 (i.e., v has two out-going edges, a continuation edge and a future edge to the first node of t_2), then we call t_1 the *parent thread* of t_2, t_2 the *future thread* (of t_1) at v, and v the *fork* of t_2. A thread t_3 is a *descendant*

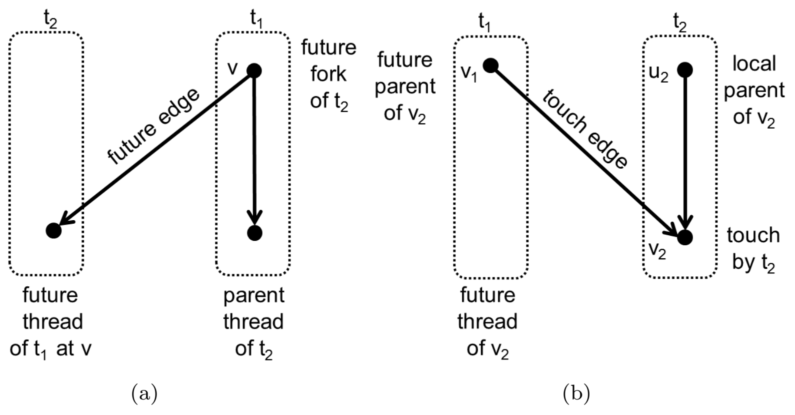


Figure 2.1: Node and thread terminology

thread of t_1 if t_3 is a future thread of t_1 or, by induction, t_3's parent thread is a descendant thread of t_1.

If there is a touch edge directed from node v_1 in thread t_1 to node v_2 in thread t_2 (i.e., t_2 touches a future computed by t_1), and a continuation edge directed from node u_2 in t_2 to v_2, then we call node v_2 a *touch of* t_1 by t_2, v_1 the *future parent* of v_2, u_2 the *local parent* of v_2, and t_1 the future thread of v_2. (Note that the touch v_2 is actually a node in thread t_2.) We call the fork of t_1 the *corresponding fork* of v_2.

Note that only touch nodes have in-degree 2. To distinguish between the two types of nodes with out-degree 2, forks and future parents of touches, we follow the convention of previous work that the children of a fork both have in-degree 1 and cannot be touches. In this way, a fork node has two children with in-degree 1, while a touch's future parent has a (touch) child with in-degree 2.

We follow the convention that when a fork appears in a DAG, the future thread is shown on the left, and the future parent on the right. (Note that this does not mean the future thread is chosen to execute first at a fork.) Similarly, the future parent of a touch is shown on the left, and the local parent on the right.

We use the following (standard) notation. Given a computation DAG, P is the number of processors executing the computation, t is the number of touches in the DAG, T_∞, the *computation span* (or *critical path*), is the length of the longest directed path, and C is the number of cache lines in each processor.

2.2 Work-Stealing and Cache Locality

In the paper, we focus on parsimonious work stealing algorithms [7], which have been extensively studied [7, 20, 1, 73, 19] and used in systems such as Cilk [18]. In a parsimonious work stealing algorithm, each processor is assigned a double-ended queue (deque). After a processor executes a node with out-degree 1, it continues to execute the next node if the next node is ready to execute. After the processor executes a fork, it pushes one child of the fork onto the bottom of its deque and executes the other. When the processor runs out of nodes to execute, it pops the first node from the bottom of its deque if the deque is not empty. If, however, its deque is empty, it steals a node from the top of the deque of an arbitrary processor.

In our model, a cache is fully associative and consists of multiple *cache lines*, each of which holds the data in a *memory block*. Each instruction can access only one memory block. In our analysis we focus only on the widely-used *least-recently used* (LRU) cache replacement policy, but our results about the upper bounds on cache overheads should apply to all *simple* cache replacement policies [1]. [1]

The *cache locality* of an execution is measured by the number of cache misses it incurs, which depends on the structure of the computation. To measure the effect on cache locality of parallelism, it is common to compare cache misses encountered in a sequential execution to the cache misses encountered in various parallel executions, focusing on the number of *additional* cache misses introduced by parallelism.

Scheduling choices at forks affect the cache locality of executions with work stealing. After executing a fork, a processor picks one of the two child nodes to execute and pushes the other into its deque. For a sequential execution, whether a choice results in a better cache performance is a characteristic of the computation itself. For a parallel execution of a computation satisfying certain properties, however, we will show that choosing future threads (the left children) at forks to execute first guarantees a relatively good upper bound on the number of additional cache misses, compared to a sequential execution that also chooses future threads first. In contrast, choosing the parent threads (the right children) to execute first can result in a large number of additional cache misses, compared to a sequential execution that also chooses parent threads first.

2.3 Structured Computations

Consider a sequential execution where node v_1 is executed immediately before node v_2. A *deviation* [73], also called a drifted node [1], occurs in a parallel execution if a processor P executes v_2, but not immediately after v_1. For example, p might execute v_1 after v_2, it might execute other nodes between v_1 and v_2, or v_1 and v_2 might be executed by distinct processors.

[73] showed that a parallel execution of a future-parallel computation with work stealing can incur $\Omega(PT_\infty + tT_\infty)$ deviations. This implies a parallel execution of a future-parallel computation with work stealing can incur $\Omega(PT_\infty + tT_\infty)$ additional cache misses. With minor modifications in that computation (see Figure 2.2), a parallel execution can even incur $\Omega(CPT_\infty + CtT_\infty)$ additional cache misses.

Our contribution in this paper is based on the observation that such poor cache locality occurs primarily when futures in the DAG can be touched by arbitrary threads, resulting in unrealistic and complicated dependencies. For example, in the worst-case DAGs in [73] that can incur significantly

[1]That is because the upper bounds in this paper are based on the results of [1] that bound the number of drifted nodes (i.e., deviations), and those results hold for all simple cache replacement policies, even with set associative caches, as discussed in [1].

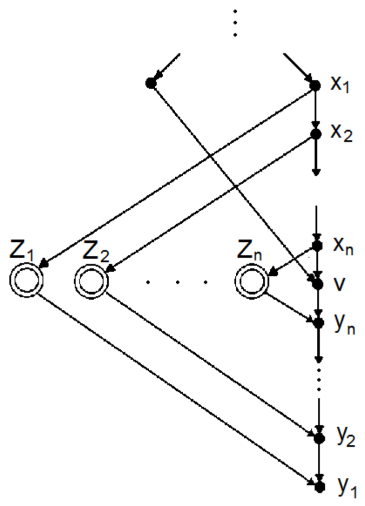


Figure 2.2: The interesting part of the bound is $\Omega(CtT_\infty)$. Figure 5 in [73] shows a DAG, as a building block of a worst-case computation, that can incur $\Omega(T_\infty)$ deviations because of one touch. We can replace it with the DAG in Figure 2.2, which can incur $\Omega(CT_\infty)$ additional cache misses due to one touch v (if the processor at a fork always chooses the parent thread to execute first), so that the worst-case computation in [73] can incur $\Omega(CtT_\infty)$ additional cache misses because of t such touches. This DAG is similar to the DAG in Figure 2.7(a) in this paper. The proof of Theorem 10 shows how a parallel execution of this DAG incurs $\Omega(CT_\infty)$ additional cache misses.

high cache overheads, futures are touched by threads that can be created before the future threads computing these futures were created. As illustrated in Figure 2.3, a parallel execution of such a computation can arrive at a scenario where a thread touches a future before the future thread computing that future has been spawned. (As a practical matter, an implementation must ensure that such a touch does not return a reference to a memory location that has not yet been allocated.) Such scenarios are avoided by *structured* future-parallel computations (e.g. Figure 2.4) that follow certain simple restrictions.

Definition 1 *A DAG is a* structured future-parallel computation *if, (1) for the future thread t of any fork v, the local parents of the touches of t are descendants of v, and (2) at least one touch of t is a descendant of the right child of v.*

There are two reasons we require that at least one touch of t is a descendant of the right child of v. First, it is natural that a computation spawns a future thread to compute a future because the computation itself later needs that value. At the fork v, the parent thread (the right child of v) represents the "main body" of the computation. Hence, the future will usually be touched either by the parent thread, or by threads spawned directly or indirectly by the parent thread.

Second, a computation usually needs a kind of "barrier" synchronization to deal with resource release at the end of the computation. Some node in the future thread t, usually the last node, should have an outgoing edge pointing to the "main body" of the computation to tell the main body that the future thread has finished. Without such synchronization, t and its descendants will be

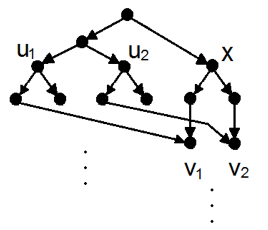


Figure 2.3: A simplified version of the DAG in [73] that can incur high cache overhead. Here, v_1 and v_2 are touches. Suppose a processor p_1 executes the root node, pushes the right child x of the root node into its deque, and then falls asleep. Now another processor p_2 steals x from p_1's deque and executes the subgraph rooted at x. Thus, v_1 and v_2 will be checked (to see if they are available) even before the corresponding future threads are spawned at u_1 and u_2.

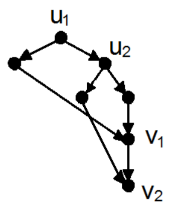


Figure 2.4: In this structured (single-touch) computation, the touches v_1 and v_2 will not be checked until their corresponding future threads have been spawned at u_1 and u_2, respectively.

isolated from the main body of the computation, and we can imagine a dangerous scenario where the main body of the computation finishes and releases its resources while t or its descendant threads are still running.

In our DAG model, such a synchronization point is by definition a touch node, though it may not be a real touch. We follow the convention that the thread that spawns a future thread releases it, so the synchronization point is a node in the parent thread or one of its descendants. Another possibility is to place the synchronization point at the last node of the entire computation, which is the typically case in languages such as Java, where the main thread of a program is in charge of releasing resources for the entire computation. These two styles are essentially equivalent, and should have almost the same bounds on cache overheads. We will briefly discuss this issue in Section 2.5.2.

We consider how the following constraint affects cache locality.

Definition 2 *A structured* single-touch *computation is a structured future-parallel computation where each future thread spawned at a fork v is touched only once, and the touch node is a descendant of v's right child.*

By the definition of threads, the future parent of the only touch of a future thread must be the last node of the future thread (the last node can also be a parent of a join node, but we don't distinguish between a touch node and a join node). The DAG in Figure 2.4 represents a structured single-touch computation. We will show that work-stealing parallel executions of structured single-touch computations achieve significantly less cache overheads than unstructured computations.

In principle, a future could be touched multiple times by different threads, so structured single-touch computations are more restrictive structured computations in general. Nevertheless, the single-touch constraint is one that is likely to be observed by many programs. For example, as noted, the Cilk [18] language supports fork-join parallelism, a strict subset of the future-parallelism model considered here. If we interpret the Cilk language's `spawn` statement as creating a future, and its `sync` statement as touching all untouched futures previously created by that thread, then Cilk programs (like all fork-join programs) are structured single-touch computations.

Structured single-touch computations encompass fork-join computations, but are strictly more flexible. Figure 2.5 presents two examples that illustrate the differences. If a thread creates multiple futures first and touches them later, fork-join parallelism requires they be touched (evaluated) in the reverse order. MethodA in Figure 2.5(a) shows the only order in which a thread can first create two futures and then touch them in a fork-join computation. This rules out, for instance, a program where a thread creates a sequence of futures, stores them in a priority queue, and evaluates them in some priority order. In contrast, our structured computations permit such futures to be evaluated by their creating thread or its descendants in any order.

Also, unlike fork-join parallelism, our notion of structured computation permits a thread to pass a future to another thread which touches that future, as illustrated in Figure 2.5(b): after a future is created, the future can be passed, as an argument of a new method call or the return value of the current thread's method call, to another thread. The thread receiving the future (MethodC in the figure) can even pass it to another thread, and so on. The only constraint is that only one of the threads that have received the future can touch it. In a fork-join computation, however, only the thread creating the future can touch it, which is much more restrictive. We believe these restrictions are easy to follow and should be compatible with how many people program in practice.

[16] observe that if a future can be touched multiple times, then complex and potentially inefficient operations and data structures are needed to correctly resume the suspended threads that are waiting for the touch. By contrast, the run-time support for futures can be significantly simplified if each future is touched at most once.

We also consider the following structured local-touch computations in the paper.

Definition 3 *A structured* local-touch *computation is one where each future thread spawned at a fork v is touched only at nodes in its parent thread, and these touches are descendants of the right child of v.*

Informally, the local touch constraint implies that a thread that needs the value of a future should create the future itself. Note that in a structured computation with local touch constraint, a future thread is now allowed to evaluate multiple futures and these futures can be touched at different times. Though allowing a future thread to compute multiple futures is not very common, [16] point out that it can be useful for some future-parallel computations like pipeline parallelism [16, 18, 38, 35, 56]. We will show in Section 2.5.1 that work-stealing parallel executions of computations satisfying the local touch constraint also have relatively low cache overheads. Note that structured computations with both single touch and local touch constraints are still a superset of fork-join computations.

```
void MethodA {
   Future x = some computation;
   Future y = some computation;
   a = y.touch();
   b = x.touch();
}
```

(a)

```
void MethodB {
   Future x = some computation;
   Future y = MethodC(x);
   ......
}
void MethodC(Future f){
    a = f.touch();
}
```

(b)

Figure 2.5: Two examples illustrating single-touch computations are more flexible than fork-join computations

2.4 Structured Single-Touch Computations

2.4.1 Future Thread First at Each Fork

We now analyze cache performance of work stealing on parallel executions of structured single-touch computations. We will show that work stealing has relatively low cache overhead if the processor at a fork always chooses the future thread to execute first, and puts the parent future into its deque. For brevity, all the arguments and results in this section assume that every execution chooses the future thread at a fork to execute first.

Lemma 4 *In the sequential execution of a structured single-touch computation, any touch x's future parent is executed before x's local parent, and the right child of x's corresponding fork v immediately follows x's future parent.*

Proof. By induction. Given a DAG, initially let S be an empty set and T the set of all touches. Note that

$$S \cap T = \emptyset \text{ and } S \cup T = \{\text{all touches}\}. \tag{2.1}$$

Consider any touch x in T, such that x has no ancestors in T. (That is, x has no ancestor nodes that are also touches.) Let t be the future thread of x and v the corresponding fork. Note that x's future parent is the last node of t by definition. When the single processor executes v, the processor pushes v's right child into the deque and continues to execute thread t. By hypothesis, there are no touches by t, since any touch by t must be an ancestor of x. There may be some forks in t. However, whenever the single processor executes a fork in t, it pushes the right child of that fork, which is a node in t, into the deque and hence t (i.e., a node in t) is right below v's right child in the deque. Therefore, the processor will always resume thread t before the right child of v. Since there is no touch by t, all the nodes in t are ready to execute one by one. Thus, when the future parent of the touch x is executed eventually, the right child of v is right at the bottom of the deque. By the single touch constraint, the local parent of x is a descendant of the right child of v, so the local parent of x cannot be executed yet. Thus, the processor will now pop the right child of v out from the bottom of the deque. Since this node is not a touch, it is ready to execute. Therefore, x satisfies the following two properties.

Property 5 *Its future parent is executed before its local parent.*

Property 6 *The right child of its corresponding fork immediately follows its future parent.*

Now set $S = S \cup \{x\}$ and $T = T - \{x\}$. Thus, all touches in S satisfy Properties 5 and 6. Note that Equation 2.1 still holds.

Now suppose that at some point all nodes in S satisfy Properties 5 and 6, and that Equation 2.1 holds. Again, we now consider a touch x in T, such that no touches in T are ancestors of x, i.e., all the touches that are ancestors of x are in S. Since the computation graph is a DAG, there must be such an x as long as T is not empty. Let t be the future thread of x and v the corresponding fork. If there are no touches by t, then we can prove x satisfies Properties 5 and 6, by the same argument for the first touch added into S. Now assume there are touches by t. Since those touches are ancestors of x, they are all in S and hence they all satisfy Property 5. When the processor executes v, it pushes v's right child into the deque and starts executing t. Similar to what we showed above, when the processor gets to a fork in t, it will always push t into its deque, right below the right child of v. Thus, the processor will always resume t before the right child of v. When the processor gets to the local parent of a touch by t, we know the future parent of the touch has already been executed since the touch satisfies Property 5. Thus, the processor can immediately execute that touch and continue to execute t. Therefore, the processor will eventually execute the future parent of x while the right child of t is still the next node to pop in the deque. Again, since the local parent of x is a descendant of the right child of v, the local parent of x as well as x cannot be executed yet. Therefore, the processor will now pop the right child of v to execute, and hence x satisfies Properties 5 and 6. Now we set $S = S \cup \{x\}$ and $T = T - \{x\}$. Therefore, all touches in S satisfy Properties 5 and 6, and Equation2.1) also holds. By induction, we have $S = \{\text{all touches}\}$ and all touches satisfy Properties 5 and 6. □

[1] have shown that the number of additional cache misses in a work-stealing parallel computation is bounded by the product of the number of deviations and the number of cache lines. It is easy to see that only two types of nodes in a DAG can be deviations: the touches and the child nodes of

forks that are not chosen to execute first. Since we assume the future thread (left child) at a fork is always executed first, only the right children of forks can be deviations. Next, we bound the number of deviations incurred by a work-stealing parallel execution to bound its cache overhead.

Lemma 7 *Let t be the future thread at a fork v in a structured single-touch computation. If t's touch x or v's right child u is a deviation, then either u is stolen or there is a touch by t which is a deviation.*

Proof. By Lemma 4, a touch is a deviation if and only if its local parent is executed before its future parent. Now suppose a processor p executes v and pushes u into its deque. Assume that u is not stolen and no touches by t are deviations. Thus, u will stay in p's deque until p pops it out. The proof of this lemma is similar to that of Lemma 4. After p spawns thread t at v, it moves to execute t. When p executes "ordinary" nodes in t, no nodes are pushed into or popped out of p's deque and hence u is still the next node in the deque to pop. When p executes a fork in t, it pushes t (more specifically, the right child of that fork) into its deque, right below u. Since a thief processor always steals from the top of a deque, and by hypothesis u is not stolen, t cannot be stolen. Thus, p will always resume t before u and then u will become the next node in the deque to pop. When p executes the local parent of a touch by t, the future parent of that touch must have been executed, since we assume that touch is not a deviation. Thus, p can continue to execute that touch immediately and keep moving on in t with its deque unchanged. Therefore, p will finally get to the local parent of x and then pop u out from its deque, since x is a descendant of u and x cannot be execute yet. Hence, neither x nor u can be a deviation. □

Theorem 8 *If, at each fork, the future thread is chosen to execute first, then a parallel execution with work stealing incurs $O(PT_\infty^2)$ deviations and $O(CPT_\infty^2)$ additional cache misses in expectation on a structured single-touch computation, where (as usual) P is the number of processors involved in this computation, T_∞ is the computation span, and C is the number of cache lines.*

Proof. [7] have shown that in a parallel execution with work stealing, there are in expectation $O(PT_\infty)$ steals. Now let us count how many deviations these steals can incur. A steal on the right child u of a fork v can make u and v's corresponding touch x_1 deviations. Suppose x_1 is a touch by a thread t_2, then the right child of the fork of t_2 and t_2's touch x_2 can be deviations. If x_2 is a deviation and x_2 is a touch by another thread t_3, then the right child of the fork of t_3 and t_3's touch x_3 can be deviation too. Note that x_2 is a descendant of x_1 and x_3 is a descendant of x_2. By repeating this observation, we can find a chain of touches $x_1, x_2, x_3, ..., x_n$, called a *deviation chain*, such that each x_i and the right child of the corresponding fork of x_i can be deviations. Since for each $i > 1$, x_i is a descendant of x_2, $x_1, x_2, x_3, \ldots, x_n$ is in a directed path in the computation DAG. Since the length of any path is at most T_∞, we have $n \leq T_\infty$. Since each future thread has only one touch, there is only one deviation chain for a steal. Since there are $O(PT_\infty)$ steals in expectation in a parallel execution [7], we can find in expectation $O(PT_\infty)$ deviation chains and in total $O(PT_\infty^2)$ touches and right children of the corresponding forks involved, i.e., $O(PT_\infty^2)$ deviations involved.

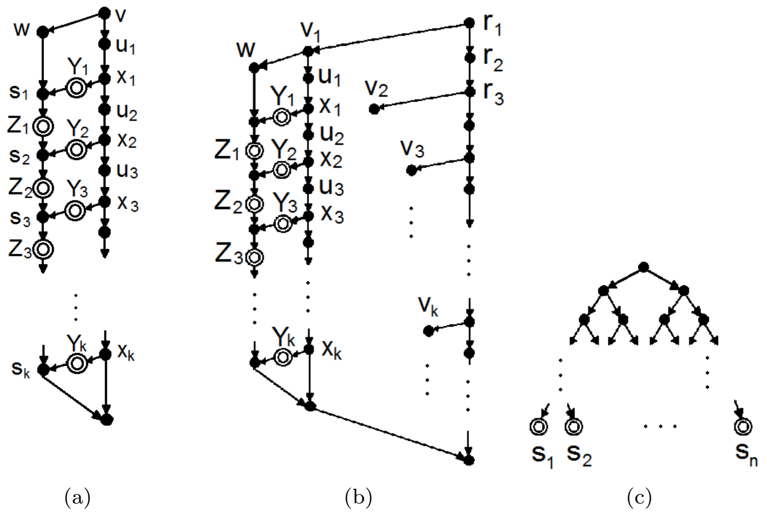


Figure 2.6: Figure (c) shows a DAG on which work stealing can incur $\Omega(PT_\infty^2)$ deviations and $\Omega(PT_\infty^2)$ additional cache misses. It uses the DAGs in (a) and (b) as building blocks.

Next, we prove by contradiction that no other touches or right children of forks can be deviations. suppose there is touch y, such that y or the right child of the corresponding fork of y is a deviation, and that y is not in any deviation chain. The right child of the corresponding fork of y can not be stolen, since by hypothesis y is not the first touch in any of those chains. Thus by Lemma 7, there is a touch y' by the future thread of y and y' is a deviation. Note that $y's$ cannot be in any deviation chain either. Otherwise y and the deviation chain y' is in will form a deviation chain too, a contradiction. Therefore, by repeating such "tracing back", we will end up at a deviation touch that is not in any deviation chain and has no touches as its ancestors. Therefore, there are no touches by the future thread of this touch, and the right child of the corresponding future fork of it is not stolen, contradicting Lemma 7.

The upper bound on the expected number of additional cache misses follows from the result of [1] that the number of additional cache misses in a work-stealing parallel computation is bounded by the product of the number of deviations and the number of cache lines. □

The bound on the number of deviations in Theorem 8 is tight, and the bound on the number of additional cache misses is tight within a factor of C, as shown below in Theorem 9.

Theorem 9 *If, at each fork node, the future thread is chosen to execute first, then a parallel execution with work stealing can incur $\Omega(PT_\infty^2)$ deviations and $\Omega(PT_\infty^2)$ additional cache misses on a structured single-touch computation, while the sequential execution of this computation incurs $O(PT_\infty^2/C)$ cache misses.*

Proof. Figure 2.6(c) shows a computation DAG on which we can get the bounds we want to prove. The DAG in Figure 2.6(c) uses the DAGs in Figures 2.6(a) and 2.6(b) as building blocks. Let's look at Figures 2.6(a) first. Suppose there are two processors p_1 and p_2 executing the DAG in

Figure 2.6(a). Suppose p_2 executes v, pushes u_1 into its deque, and then falls asleep before executing w. Now suppose p_1 steals u_1. For each $i \leq k$, neither s_i nor Z_i can be executed since w has not been executed yet. Now p_1 takes a solo run, executing $u_1, x_1, Y_1, u_2, x_2, Y_2, ..., x_k, Y_k$. After p_1 finishes, p_2 wakes up and executes the rest of the computation DAG. Note that the right (local) parent of s_i is executed before the left (future) parent of the touch is executed. Thus, by Lemma 4, each s_i is a deviation. Hence, this parallel execution incurs k deviations and the computation span of the computation is $\Theta(k)$.

Now let us consider a parallel execution of the computation in 2.6(b). For each $i \leq k$, the subgraph rooted at v_i is identical to the computation DAG in 2.6(a) (except that the last node of the subgraph has an extra edge pointing to a node of the main thread). Suppose there are three processors p_1, p_2, and p_3 working on the computation. Assume p_2 executes r_1 and v_1 and then falls asleep when it is about to execute w. p_3 now steals r_2 from p_2 and then falls asleep too. Then p_1 steals u_1 from p_2's deque. Now p_1 and p_2 execute the subgraph rooted at v_1 in the same way they execute the DAG in 2.6(a). After p_1 and p_2 finish, p_3 wakes up, executes r_2. Now these three processors start working on the subgraph rooted at r_3 in the same way they executed the graph rooted at r_1. By repeating this, the execution ends up incurring k^2 deviations when all the k subgraphs are done. Since the length of the path $r_1, r_2, r_3...$ on the right-hand side is $\Theta(k)$, the computation span of the DAG is still $\Theta(k)$.

Now we construct the final computation DAG, as in Figure 2.6(c). The "top" nodes of the DAG are all forks, each spawning a future thread. Thus, they form a binary tree and the number of threads increase exponentially. The DAG stops creating new threads at level $\Theta(\log n)$ when it has n threads rooted at $S_1, S_2, ..., S_n$, respectively. For each i, the subgraph rooted at S_i is identical to the DAG in 2.6(b). Suppose there are $3n$ processors working on the computation. It is easy to see n processors can eventually get to $S_1, S_2, ..., S_n$. Suppose they all fall asleep immediately after executing the first two nodes of S_i(corresponding to r_1 and v_1 in Figure 2.6(b)) and then each two of the rest $2n$ free processors join to work on the subgraph rooted at S_i, in the same way p_1, p_2 and p_3 did in Figure 2.6(b). Therefore, this execution will finally incur nk^2 deviations, while the computation span of the DAG is $\Theta(k + \log n)$. Therefore, by setting $n = P/3$, we get a parallel execution that incurs $\Omega(PT_\infty^2)$ deviations, when $\log P = O(k)$.

To get the bound on the number of additional cache misses, we just need to modify the graph in 2.6(a) as follows. For each $1 \leq i \leq k$, Y_i consists of a chain of C nodes $y_{i1}, y_{i2}, ..., y_{iC}$, where C is the number of cache lines. $y_{i1}, y_{i2}, ..., y_{iC}$ access memory blocks $m_1, m_2, ..., m_C$, respectively. Similarly, each Z_i consists of a chain of C nodes $z_{i1}, z_{i2}, ..., z_{iC}$. $z_{i1}, z_{i2}, ..., z_{iC}$ access memory blocks $m_C, m_{C-1}, ..., m_1$, respectively. all s_i access memory block m_C. For all $1 \leq i \leq k$, u_i and x_i both access memory block m_{C+1}. It does not matter which memory blocks the other nodes in the DAG access. For simplicity, assume the other nodes do not access memory. In the sequential execution, the single processor has $m_1, m_2, ..., m_C$ in its cache after executing v, w, u_1, x_1, Y_1, Z_1 and it has incurred $(C+1)$ cache misses so far. Now it executes u_2 and x_2, incurring one cache miss at node u_2 by replacing m_C with m_{C+1} in its cache, since m_C is the least recently used block. When it executes

Y_2 and Z_2, it only incurs one cache miss by replacing m_{C+1} with m_C at the last node of Y_2, y_{2C}. Likewise, it is easy to see that the sequential execution will only incur cache misses at nodes u_i and at the last nodes of Y_i for all i. Hence, the sequential execution incurs only $O(k+C)$ cache misses. When $k = \Omega(C)$, the sequential execution incurs only $O(k)$ cache misses.

Now consider the parallel execution by two processors p_1 and p_2 we described before. p_2 will incur only C cache misses, since Z_i and s_i only access m different blocks $m_1, m_2, ..., m_C$ and hence p_2 doesn't need to swap any memory blocks out of its cache. However, p_1 will incur lots of cache misses. After executing each Y_i, p_1 will execute u_{i+1}. Thus at u_{i+1}, one cache miss is incurred and m_1 is replaced with m_{C+1}, since m_1 is the least recently used block. Then, when p_1 executes the first node $y_{(i+1)1}$ in Y_i, , m_1 is not in its cache. Since m_2 now becomes the least recently used memory block in p_1's cache, m_2 is replaced by m_1. Thus, m_2 will not be in the cache when it is in need at $y_{(i+1)2}$. Therefore, it is obvious that p_1 will incur a cache miss at each node in Y_i and hence incur Ck cache misses in total in the entire execution. Note that the computation span of this modified DAG is $\Theta(Ck)$, since each Z_i now has C nodes. Therefore, the sequential execution and the parallel execution actually incur $\Theta(T_\infty/C)$ and $\Theta(T_\infty)$, respectively, when $\log P = O(k)$. Therefore, if we use this modified DAG as the building blocks in 2.6(c), we will get the bound on the number of additional cache misses stated in the theorem. $\square$

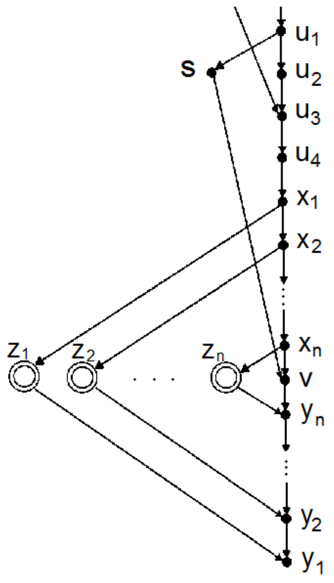
2.4.2 Parent Thread First at Each Fork

In this section, we show that if the parent thread is always executed first at a fork, a work-stealing parallel execution of a structured single-touch computation can incur $\Omega(tT_\infty)$ deviations and $\Omega(CtT_\infty)$ additional cache misses, where t is the number of touches in the computation, while the corresponding sequential execution incurs only a small number of cache misses. This bound matches the upper bound for general, unstructured future-parallel computations [73][2]. This result, combined with the result in Section 2.4.1, shows that choosing the future threads at forks to execute first achieves better cache locality for work-stealing schedulers on structured single-touch computations.

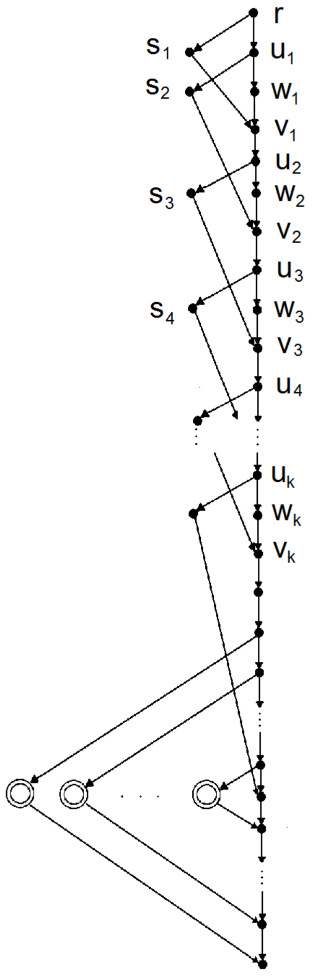
Theorem 10 *If, at each fork, the parent thread is chosen to execute first, then a parallel execution with work stealing can incur $\Omega(tT_\infty)$ deviations and $\Omega(CtT_\infty)$ additional cache misses on a structured single-touch computation, while the sequential execution of this computation incurs only $O(C+t)$ cache misses.*

Proof. The final DAG we want to construct is in Figure 2.8. It uses the DAGs in Figure 2.7 as building blocks. We first describe how a single deviation at a touch u_3 can incur $\Omega(T_\infty)$ deviations and $\Omega(CT_\infty)$ additional cache misses in Figure 2.7(a). In order to get the bound we want to prove, here we follow the convention in [1, 73] to distinguish between touches and join nodes in the DAG. More specifically, y_i is a join node, not a touch, for each $1 \leq i \leq n$. For each $1 \leq i \leq n$, node

[2]The bound on the expected number of deviations in [73] is actually $O(PT_\infty + tT_\infty)$. However, as pointed out in [73], a simple fork-join computation can get $\Omega(PT_\infty)$ deviations. Hence we focus on the more interesting part $\Omega(tT_\infty)$.



(a)



(b)

Figure 2.7: DAGs used by Figure 2.8 as building blocks.

x_i accesses memory block m_1 and y_i accesses memory block m_{C+1}. Z_i consists of a chain of C nodes $z_{i1}, z_{i2}, ..., z_{iC}$, accessing memory blocks $m_1, m_2, ..., m_C$ respectively. All the other nodes do not access memory. Assume in the sequential execution a single processor p_1 executes the entire DAG in Figure 2.7(a). Suppose initially the left (future) parent of u_3 has already been executed. p_1 starts executing the DAG at u_1. Since p_1 always stays on the parent thread at a fork, it first pushes s into its deque, continues to execute u_2, u_3, u_4, and then executes $x_1, x_2, ..., x_n$ while pushing $z_{11}, z_{21}, ..., z_{n1}$ into its deque. Since v cannot be executed due to s, p_1 pops z_{n1} out of its deque and executes the nodes in Z_n. Then p_1 executes all the nodes in $Z_{n-1}, Z_{n-2}, ..., Z_1$, in this order. So far p_1 has only incurred C cache misses, since all the nodes it has executed only access memory blocks $m_1, ..., m_C$ and hence it did not need to swap any memory blocks out of its cache. Now p_1 executes s, v and then $y_n, y_{n-1}, ..., y_1$, incurring only one more cache miss by replacing m_1 with m_{C+1} at y_n. Hence, this execution incurs $O(C)$ cache misses in total. Note that the left parent of y_i is executed before the right parent y_i for all i.

Now assume in another execution by p_1, the left parent of u_3 is in p_1's deque when p_1 starts executing u_1. Thus, u_3 is a deviation with respect to the previous execution. Since u_3 is not ready to execute after p_1 executes u_2, p_1 pops s out of its deque to execute. Since v is not ready, p_1 now pops the left parent of u_3 to execute and then executes $u_3, u_4, x_1, x_2, ..., x_n, v$. Now p_1 pops z_{n1} out and executes all the nodes Z_n. Note that y_n is now ready to execute and the memory blocks in p_1's cache at the moment are $m_1, m_2, ..., m_C$. Now p_1 executes y_n, replacing the least recently used block m_1 with m_{C+1}. p_1 then pops $z_{(n-1)1}$ out and executes all the nodes $z_{(n-1)1}, z_{(n-1)2}, ..., z_{(n-1)}C$ in Z_{n-1} one by one. When p_1 executes $z_{(n-1)1}$, it replaces m_2 with m_1, and when it executes $z_{(n-1)2}$, it replaces m_3 with m_2, and so on. The same thing happens to all Z_i and y_i. Thus, p_1 will incur a cache miss at every node afterwards, ending up with $\Omega(Cn)$ cache misses in total. Note that the computation span of this DAG is $T_\infty = \Theta(C+n)$. Thus, this execution with a deviation at u_3 incurs $\Omega(CT_\infty)$ cache misses when $n = \Omega(C)$. Moreover, all y_i are deviations and hence this execution incurs $\Omega(T_\infty)$ deviations.

Now let us see how a single steal at the beginning of a thread results in $\Omega(T_\infty)$ deviations and $\Omega(CT_\infty)$ cache misses at the end of the thread. Figure 2.7(b) presents such a computation. First we consider the sequential execution by a processor p_1. It is easy to see p_1 executes nodes in the order $r, u_1, w_1, s_2,\ s_1, v_1, u_2,\ w_2, v_2, u_3, w_3,\ s_4, s_3, v_3, u_4, ...$ The key observation is that w_i is executed before s_i is executed for any odd-numbered i while w_i is executed after s_i is executed for any even-numbered i. This statement can be proved by induction. Obviously, this holds for $i = 1$ and $i = 2$, as we showed before. Now suppose this fact holds for all $1, 2, ..., i$, for some even-numbered i. Now suppose p_1 executes u_{i-1}. Then p_1 pushes s_i into its deque and executes w_{i-1}. Since we know w_{i-1} should be executed before s_{i-1}, s_{i-1} has not been executed yet. Moreover, s_{i-1} must already be in the deque before s_i was pushed into the deque, since s_{i-1}'s parent u_{i-2} has been executed and s_{i-1} is ready to execute. Now p_1 pops s_i out to execute. Since v_i is not ready to execute, p_1 pops s_{i-1} out and then executes s_{i-1}, v_{i-1}, u_i, and pushes s_{i+1} into the deque. Now p_1 continues to execute w_i, v_i, u_{i+1} and pushes s_{i+1} into its deque. Then p_i executes w_{i+1} and pops s_{i+2} out, since v_{i+1} is not ready due to s_{i+1}. Now we can see w_{i+1} and s_{i+2} have been executed, but s_{i+1} and w_{i+2} not yet. That is, w_{i+1} is executed before s_{i+1} and w_{i+2} is executed after s_{i+2}. Therefore, the statement holds for $i+1$ and $i+2$, and hence the proof completes.

The subgraph rooted at u_k is identical to the graph in Figure 2.7(a), with v_k corresponding to u_3 in Figure 2.7(a). Therefore, if k is an even number, v_k's left parent has been executed when w_k is executed and hence the sequential execution will incur only $O(C)$ cache misses on the subgraph rooted at u_k.

Now consider the following parallel execution of the DAG in Figure 2.7(b) by two processors p_1 and p_2. p_1 first executes r and pushes s_1 into its deque. Then p_2 immediately steals s_1 and executes it. Now p_2 falls asleep, leaving p_1 executing the rest of the DAG alone. It is easy to see p_1 will execute the nodes in the DAG in the order $u_1, w_1, v_1, u_2, w_2, s_3, s_2, v_2, u_3, w_3, v_3, u_4, s_4, ...$ It can be proved by induction that w_i is executed after s_i is executed for any odd-numbered i while w_i is executed before s_i is executed for any even-numbered i, which is opposite to the order in the sequential execution. The induction proof is similar to that of the previous observation in the sequential execution, so we omit the proof here. If k is an even number, w_k will be executed before the left parent of v_k and hence this execution will incur $\Omega(T_\infty)$ deviations and $\Omega(CT_\infty)$ cache misses when $n = \Omega(C)$ and $n = \Omega(k)$.

The final DAG we want to construct is in Figure 2.8. This is actually a generalization of the DAG in Figure 2.7(b). Instead of having one fork u_i before each touch v_i, it has two forks u_i and x_i, for each i. After each touch v_i, the thread at y_i splits into two identical branches, touching the futures spawned at u_i and x_i, respectively. In this figure, we only depict the right branch and omit the identical left branch. As we can see, the right branch later has a touch v_{i+1} touching the future s_{i+1} spawned at the fork x_i. If we only look at the thread on the right-hand side, it is essentially the same as the DAG in Figure2.7(b). The sequential execution of this DAG by p_1 is similar to that in Figure2.7(b). The only difference is that p_1 at each y_i will execute the right branch first and then the left branch recursively. Similarly, it can be proved by induction that w_i is executed before s_i is executed for any odd-numbered i while w_i is executed after s_i is executed for any even-numbered i. Obviously this also holds for each left branch. Now consider a parallel execution by two processors p_1 and p_2. p_1 first executes r. p_2 immediately steals s_1 and executes it and then sleeps forever. Now p_1 makes a solo run to execute the rest of the DAG. Again, we can prove by the same induction argument that w_i is executed after s_i is executed for any odd-numbered i while w_i is executed before s_i is executed for any even-numbered i, which is opposite to the order in the sequential execution. The above two induction proofs are a little more complicated than those for the DAG in Figure2.7(b), but the ideas are essentially the same (the only difference is now we have to prove the statements hold for the two identical branches split at fork y_i at the inductive step) and hence we omit the proofs again.

By splitting each thread into two after each y_i, the number of branches in the DAG increases exponentially. Suppose there are t touches in the DAG. Thus, there are eventually $\Theta(t)$ branches and the height of this structure is $\Theta(\log t)$. At the end of each branch is a subgraph identical to the

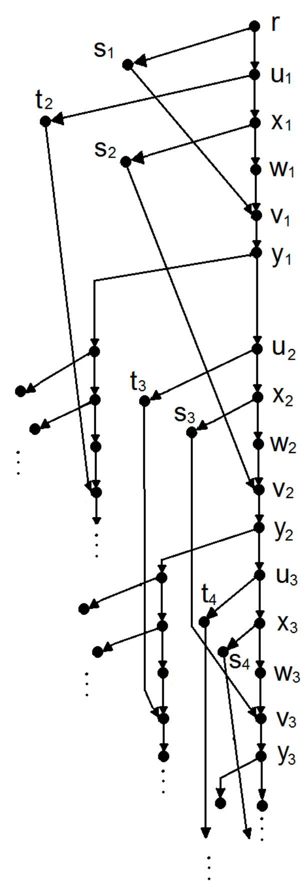


Figure 2.8: A DAG on which work stealing can incur $\Omega(tT_\infty)$ deviations and $\Omega(CtT_\infty)$ if it chooses parents threads to execute first at forks. This example uses the DAGs in Figure 2.7 as building blocks.

DAG in Figure 2.7(a). Therefore, the parallel execution with only one steal can end up incurring $\Theta(tn)$ deviations and $\Theta(Ctn)$ cache misses. The sequential execution incurs only $\Theta(C + t)$ cache misses, since the sequential execution will incur only 2 cache misses by swapping m_{C+1} in and out at each branch, after it incurs C cache misses to load $m_1, m_2, ..., m_C$ at the first branch. hence, when $n = \Omega(\log t)$ and $n = \Omega(C)$, we get the bound stated in the theorem. □

2.5 Other Kinds of Structured Computations

It is natural to ask whether other kinds of structured computations can also achieve relatively good cache locality. We now consider two alternative kinds of restrictions.

2.5.1 Structured Local-Touch Computations

In this section, we prove that work-stealing parallel executions of structured local-touch computations also have relatively good cache locality, if the future thread is chosen to execute first at each fork. This result, combined with Theorems 8 and 10, implies that work-stealing schedulers for structured computations are likely better off choosing future threads to execute first at forks.

Lemma 11 *In the sequential execution of a structured local-touch computation where the future thread at a fork is always chosen to execute first, any touch x's future parent is executed before x's local parent, and the right child of any fork v immediately follows the last node of the future thread spawned at v, i.e., the future parent of the last touch of the future thread.*

The proof is omitted because it is almost identical to that of Lemma 4. (We first consider a future thread whose touches are the "earliest" in the DAG, that is, no other touches are ancestors of them, and we can easily prove the statement in Lemma 11 holds for those touches. Then by the same induction proof as for Lemma 4, we can prove the statement holds for all future threads' touches.)

Theorem 12 *If the future thread at a fork is always chosen to execute first, then a parallel execution with work stealing incurs $O(PT_\infty^2)$ deviations and $O(CPT_\infty^2)$ additional cache misses in expectation on a structured local-touch computation.*

Proof. Let v be a fork that spawns a future thread t. Now we consider a parallel execution. Let p be a processor that executes v and pushes the right child of v into its deque. Suppose the right child of v is not stolen. Now consider the subgraph G' consisting of t and its descendant threads. Note that G' itself is a structured computation DAG with local touch constraint. Now p starts executing G'.

According to local touch constraint, the only nodes outside G' that connect to the nodes in G' are v and the touches of t, and c is the only node outside G' that the nodes in G' depend on. Now v has been executed and the touches of t are not ready to execute due to the right child of v. Hence, p is able to make a sequential execution on G' without waiting for any node outside to be done or jumping to a node outside, as long as no one steals a node in G' from p's deque. Since we assume the right child of v will not be stolen and any nodes in G' can only be pushed into p's deque below v, no nodes in G' can be stolen. Hence, G' will be executed by a sequential execution by p. Therefore, there are no deviations in G'. After p executed the last node in G', which is the last node in t, p pops the right child of v to execute. Hence, the right child of v cannot be a deviation either, if it is not stolen. That is, those nodes can be deviations only if the right child of v is stolen. Since there are in expectation $O(PT_\infty)$ steals in an parallel execution and each future thread has at most T_∞ touches, the expected number of deviations is bounded by $O(PT_\infty^2)$ and the expected number of additional touches is bounded by $O(CPT_\infty^2)$. □

2.5.2 Structured Computations with Super Final Nodes

As discussed in Section 2.3, in languages such as Java, the program's main thread typically releases all resources at the end of an execution. To model this structure, we add an edge from the last node of each thread to the final node of the computation DAG. Thus, the final node becomes the only node with in-degree greater than 2. Since the final node is always the last to execute, simply adding those edges pointing to the final node into a DAG will not change the execution order of the nodes in the DAG. It is easy to see that having such a super node will not change the upper bound on the cache overheads of the work-stealing parallel executions of a structured computation.

For structured computations with super final nodes, it also makes sense to slightly relax the single-touch constraint as follows.

Definition 13 *A structured single-touch computation with a* super final node *is one where each future thread t at a fork v has at least one and at most* two *touches, a descendant of v's right child and the super final node.*

In such a computation, a future thread can have the super final node as its only touch. This structure corresponds to a program where one thread forks another thread to accomplish a side-effect instead of computing a value. The parent thread never touches the resulting future, but the computation as a whole cannot terminate until the forked thread completes its work.

Now we show that the parallel executions of structured single-touch computations with super final nodes also have relatively low cache overheads.

Lemma 14 *In the sequential execution of a structured single-touch computation with a super final node, where the future thread at a fork is always chosen to execute first, any touch x's future parent is executed before x's local parent, and the right child u of any fork v immediately follows the last node of the future thread spawned at v, i.e., the future parent of the last touch of the future thread.*

Lemma 15 *Let t be the future thread at a fork v in a structured single-touch computation with a super final node. If a touch of t or v's right child u is a deviation, then either u is stolen or there is a touch by t which is a deviation.*

Proof. The proofs of Lemma 4 and Lemma 7, with only minor modifications, also apply to the above two lemmas, respectively. That is because introducing the super final node into a computation doesn't affect the order in which other nodes are executed, since no other nodes need to wait for the super final node and the super final node is always the last node to execute. More specifically, when a processor executing any thread t reaches a node that is a parent of the super final node, the processor will continue to work on t if that node is not the last node of t, and otherwise try popping a node out of its deque. Therefore, by the same proof techniques as for Lemmas 4 and 7, we can show that a processor will execute the right child u of a fork v and the parents of the touches of the future spawned at v in the order stated in Lemmas 14 and 15. $\square$

Theorem 16 *If, at each fork, the future thread is chosen to execute first, then a parallel execution with work stealing incurs $O(PT_\infty^2)$ deviations and $O(CPT_\infty^2)$ additional cache misses in expectation on a structured single-touch computation with a super final node.*

Proof. The proof is similar to that of Theorem 8. The only difference is that if a touch by a thread t is a deviation, now the two touches of t can both be deviations, which could be a trouble for constructing the deviation chains. Fortunately, one of these two touches is the super final node, which is always the last node to execute and hence will not make the touches of other threads become deviations. Therefore, we can still get a unique deviation chain starting from a steal and hence the proof of Theorem 8 still applies here. □

Similarly, we can also introduce a super final node to a structured local-touch computation as follows.

Definition 17 *A structured local-touch computation with a* super final node *is one where each future thread t spawned at a fork v can be touched only by the super final node and by t's parent thread at nodes that are descendants of the right child of v.*

It is obvious that by the same proof as for Theorem 12, we can prove the following bounds.

Theorem 18 *If the future thread at a fork is always chosen to execute first, then a parallel execution with work stealing incurs $O(PT_\infty^2)$ deviations and $O(CPT_\infty^2)$ additional cache misses in expectation on a structured local-touch computation with a super final node.*

Chapter 3

Theoretical Analysis of Work Stealing on Hierarchical Platforms

In this chapter, we will discuss the performance of fork-join programs on hierarchical platforms. We present lower bounds for their executions by all work stealing algorithms as well as upper bounds for specific work stealing variants in a theoretical hierarchical model. We first discuss the related work in Section 3.1. Then in Section 3.2, we introduce the theoretical model for hierarchical systems and the computation models for fork-join and divided-and-conquer programs, as well as their s-bounded subclasses. In Section 3.3, we define the class of work stealing algorithms, and introduce the global work stealing algorithm with its attaching scheme. Section 3.4 presents upper and lower bounds of work stealing algorithms on general fork-join programs, while Section 3.5 shows their bounds on s-bounded fork-join and s-bounded devide-and-conquer programs.

3.1 Related Work

Work stealing [20] on traditional shared-memory systems has been an active research area for decades. On the theoretical side, its time and space bounds [15, 7, 19, 20] as well as its cache locality [1, 73, 47] for different parallel program models, such as fork-join parallelism (i.e., nested parallelism) [15] and future parallelism [41, 42], have been extensively studied.

To implement efficient work stealing algorithms on hierarchical platforms such as NUMA systems and distributed clusters, different heuristic strategies for choosing victim processors to steal from have been discussed. A heuristic people have found effective is that processors should be in favor of local steals if neighboring processors have enough work to be stolen, in order to reduce the number of costly remote steals. A commonly used method is to have a processor first make one or more steal attempts locally and then try remote steals only if all the local steal attempts have failed [62]. Another method is to have each processor make a local steal with a higher probability than a remote steal [70]. Other implementations [24, 66] only allow leaders of sockets/clusters to make remote steals.

Researchers have also examined the appropriate amounts of work a processor should steal. Many papers (e.g., [62, 65, 76]) suggested the StealHalf policy for remote steals, that is, a thief should steal half of the tasks from a remote processor's task pool, as opposed to stealing only one task in the classical work stealing algorithm.

Work stealing is usually implemented using concurrent deques in traditional shared-memory systems. However, in some hierarchical platforms such as distributed clusters, concurrent deques for remote steals can be very inefficient or even impossible to implement. For this reason, researchers [29, 71, 78, 2, 57] have proposed work stealing variants with non-concurrent deques via message passing.

In contrast to the large body of research on the systems side we just mentioned above, the theoretical analysis of hierarchical work steal has been much less discussed. The only theoretical bound we are aware of is given by Quintin and Wagner [70], on the time complexity of their hierarchical work stealing (HWS) algorithm. The leader processors of clusters in HWS first distribute tasks across a system, and then the worker processors in each cluster execute their local tasks. However, the time bound of HWS can be very bad in theory, since the workloads in different clusters can be highly unbalanced unless HWS knows the structure of the target program very well in advance. Therefore, the upper bound of HWS cannot help us figure out the real bounds of the class of all work stealing algorithms.

Some hierarchical cache models, such as parallel cache-oblivious (PCO) model [14], hierarchical multi-level multicore (HM) model [25], and Threaded Many-core Memory (TMM) model [61], have been proposed. They focus on the cache locality of parallel programs, while the results of this paper focus more on the costs of remote communications caused by remote steals and joins. The performance of work stealing in these models has not be studied yet, and we consider it an interesting open problem.

3.2 Fork-Join Model on Hierarchical Systems

3.2.1 Fork-Join Model

Fork-join parallelism is also called nested parallelism. As in previous work (e.g., [1] [20]), we model a program as a directed acyclic graph (DAG), where each node represents a single instruction of the program that can be executed by a processor and each directed edge (u, v) indicates that v cannot be executed until u has been executed. Both the indegree and outdegree of a node in a DAG are at most two.

There are three types of edges in the model—continuation edges, spawn edges, and join edges. At most one of the incoming edges and one of the outgoing edges of a node are *continuation edges*. A thread is represented as a maximal sequence of nodes connected by continuation edges in a DAG. Obviously, the nodes, i.e., the instructions, in a thread can only be executed sequentially.

If a node has two outgoing edges, we call it a *fork* node. One outgoing edge of a fork is a

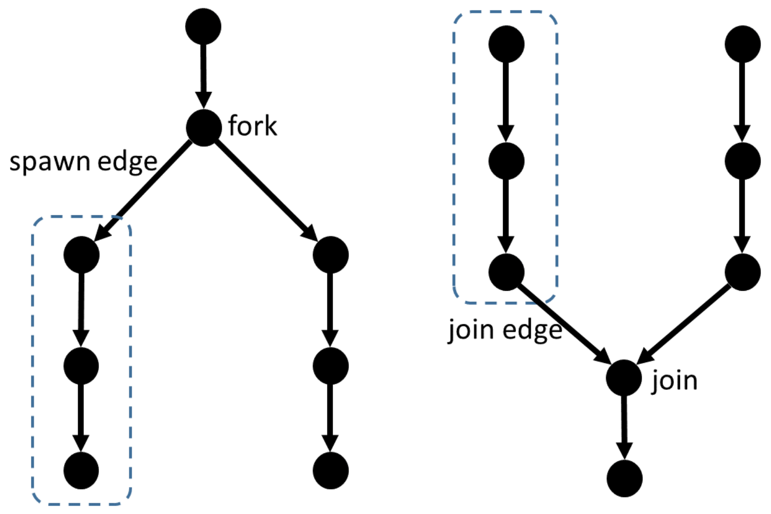


Figure 3.1: The graph on the left hand side shows a thread in the dashed box spawned by its parent thread at a fork. The graph on the right hand side shows the join of a thread in the dashed box and its parent thread.

continuation edge and the other is a *spawn edge*. A thread x spawns a new thread y at a fork node u in x: the spawn edge of u points to the first node of thread y (see Figure 3.1). We call x the *parent thread* of y, and y a *child thread* of x. For a better illustration, we always put the spawn edge of a fork to the left of its outgoing continuation edge in a figure in the paper.

If a node has two incoming edges, we call it a *join* node, representing the join of a thread and its parent thread. (In the fork-join model, only a thread and its parent thread can join, but in the more general future-parallel model [41, 42], any two threads can join in principle.) One incoming edge (on the right-hand side in a figure) of a join is a continuation edge and the other (on the left-hand side) is a *join edge* (see Figure 3.1). The node that the join edge comes from is in the child thread. This node must be the last node of the child thread, since the node has no outgoing continuation edge. (If the node has an outgoing continuation edge, it must be a fork by definition. However, a fork cannot have an outgoing spawn edge, a contradiction.) A node cannot be both a fork and a join, since it represents only a single instruction of a thread.

In a typical fork-join program in practice, when a thread wants to partition some computation into two parts to execute in parallel, it makes a fork operation, creating a child thread for one part and working on the other itself. When the thread finishes its work, it calls a join, retrieving the result of the child thread if the child thread has completed, and waiting for the child thread otherwise.

There is only one node of indegree 0 in a DAG, called the *source node*, and only one node of outdegree 0, called the *sink node*. The source node and the sink node are the first and the last nodes of the *main thread* respectively. All the other threads are spawned directly or indirectly by the main thread (see 3.2 for an example). It is easy to see the source node is the first node to execute in the DAG and the sink node is the last.

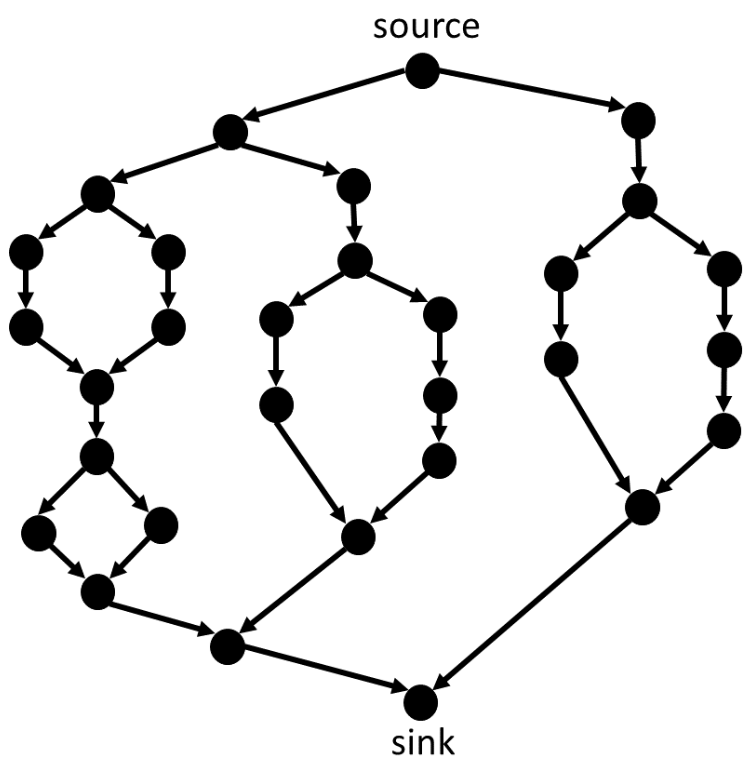


Figure 3.2: A fork-join DAG, where the main thread is the rightmost directed path in the DAG.

Another requirement for fork-join DAGs is that if a thread spawns some child threads at different forks before joining any of them, these child threads must join in a reverse order. This constraint guarantees the forks and joins of threads in a DAG are in a "nested form" and that is why fork-join programs is also called nested-parallel programs. [1]

As Acar et al. [1] pointed out, a fork-join DAG can also be defined by induction: if we only consider the nodes between a fork and its corresponding join, the two threads derived at the fork can be thought of as the main threads of two independent DAGs that eventually merge at the join.

Algorithm 1 Fibonacci(n)

```
if $n > 5$ then
    fork f1 = Fibonacci(n-1);
    f2 = Fibonacci(n-2);
    join f1;
    return f1 + f2;
else
    return Sequential_Fibonacci(n);
```

[1] Another common way to create parallel threads is by calling a parallel-for loop, where each iteration of a for-loop in a parent thread creates a parallel child thread and the parent later makes a single join call to collects the results of all child threads. In fact, a parallel-for loop can also be thought of as a fork-join DAG, if we model the single join as a sequence of consecutive joins for those child threads, in a reverse order.

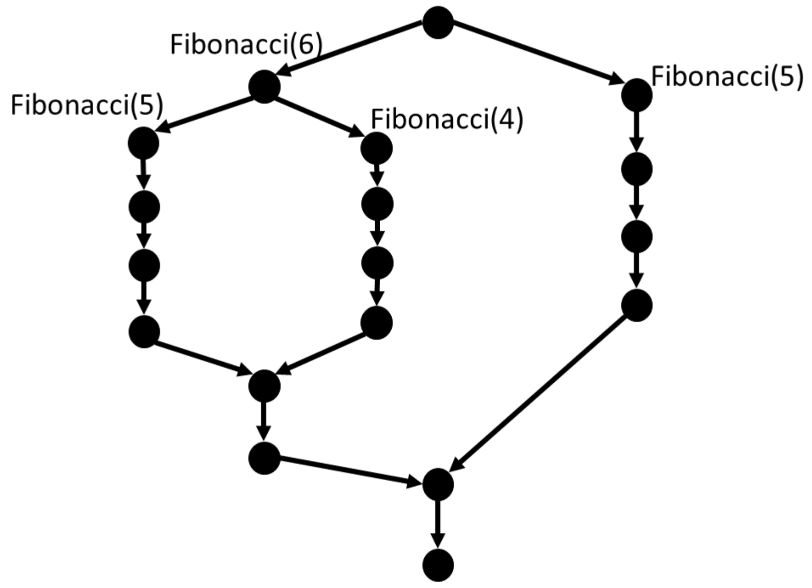


Figure 3.3: The DAG of Fibonacci(7) of Algorithm 1. The main thread (i.e., the rightmost one) spawns a child thread x to compute Fibonacci(6) and then continues to compute Fibonacci(5) itself. Thread x spawns a child thread y of its own to compute Fibonacci(5) and then computes Fibonacci(4) itself. According to the pseudocode, Fibonacci(5) and Fibonacci(4) will be executed sequentially. Later thread x sums up Fibonacci(4) and Fibonacci(5) after the join of x and y, and finally the main thread sums up Fibonacci(5) and Fibonacci(6) after the join of x and itself.

Algorithm 1 is the pseudocode for computing the nth Fibonacci number recursively and Figure 3.3 illustrates how Fibonacci(7) is modeled as a fork-join DAG.

We use T_1 to denote the total number of nodes in a DAG and T_∞ to denote the number of nodes in a longest directed path in the DAG. T_∞ is called the *critical path length* (or the span) of the DAG. Since we will focus on the theoretical bounds of DAGs with different ratios of $\frac{T_1}{T_\infty}$, we use $\mathcal{G}(T_1, T_\infty)$ to denote a DAG of T_1 nodes with critical path length T_∞.

A well-studied subset of fork-join programs are *divided-and-conquer programs*. In a divided-and-conquer program, a thread has no forks after the first join node in the thread. In other words, threads in a divided-and-conquer program first keep forking to compute things in parallel and later join in pairs recursively. Most fork-join programs discussed in the literature, such as the Merge-Sort algorithm, the N-Queens algorithm, and the Fibonacci algorithm shown in Figure 3.3, are in fact divide-and-conquer programs.

3.2.2 S-Bounded Fork-Join Programs

Prior research (e.g., [60]) has found it beneficial empirically to prevent a computation from splitting into too small tasks. This is because the extra parallelism provided by splitting a sequential computation into tiny parallel subroutines may not pay off the costs of spawning and joining threads, as well as the communication costs among processors, especially on a hierarchical system where remote operations and communications are expensive. For instance, it is common that a Merge-Sort program stops partitioning an array of numbers into smaller ones when the size of the array is a few kilobytes [70]. Another example is the recursive 8-Queens algorithm in [70]. The algorithm first spawns a thread for each feasible position of the queen on the first line. Then each thread spawns a new thread for each feasible position of the queen on the second line and so on. When a thread reaching a feasible position of the queen on the fourth line, it stops splitting and the rest of the work is executed sequentially in the thread.

To analyze the performance of work stealing on these algorithms, we define the class of *s-bounded* fork-join programs. We say a DAG is s-bounded, if each thread in it has at least s nodes. We use $\mathcal{G}(T_1, T_\infty, s)$ to denote an s-bounded DAG $\mathcal{G}(T_1, T_\infty)$.

We will show in Section 3.5 some upper and lower bounds for s-bounded fork-join programs and s-bounded divide-and-conquer programs.

3.2.3 Hierarchical System Model

The hierarchical system model we consider in the paper consists of k distributed, homogeneous clusters. Each cluster has n processors and there are in total $p = kn$ processors in the system. A processor can execute nodes in a DAG, one at a time, and communicate with other processors in order to distribute nodes across the system or share information. The cluster a processor belongs to is the local cluster and the other clusters are remote clusters to it. Processors within a cluster are local to each other while Processors in different clusters are remote to each other.

Communication is efficient in time within a cluster and inefficient across different clusters. More specifically, there is a global clock on a hierarchical system, which processors may or may not be aware of it. A tick of the global clock is the minimum unit of time. The execution of a node in a DAG, as well as one round of one-way communication between two local processors, takes time 1, while one round of one-way communication between two remote processors, called a *remote operation*, takes time r. Our communication model can be thought of as a simplified version of the LogP model [27].

We keep the communication model abstract in order to cover both hierarchical shared-memory and message-passing systems. In a hierarchical shared memory system, for instance, each cluster has its local shared memory that can be efficiently accessed by processors in the cluster: we can assume it takes a processor time 1 to make a read or write on the memory of the local cluster, while it takes time r to do so on the memory of a remote cluster. In a hierarchical message-passing model, it takes time 1 for a message sent by a processor to arrive at a local processor, while it takes time r for a message to arrive at a remote processor.

For any load balancing algorithm executing a fork-join program, there are at least two scenarios where a remote operation has to be incurred. First, if a thread executed by a processor in a cluster $\mathcal{C}_a$ spawns a new thread at a fork and later a remote processor in another cluster $\mathcal{C}_b$ wants to take the spawned thread to execute, a remote operation must be made. That is because the information about the new thread has to be transferred to $\mathcal{C}_b$, either by a message sent to $\mathcal{C}_b$ in the message-passing model, or by a remote read from $\mathcal{C}_a$ and/or a remote write to $\mathcal{C}_b$ in the shared-memory model. In either case, it takes at least time r.

Second, if a thread and its parent thread are held in two different clusters, then at least one remote operation has to be made by the time the join node of the two threads is executed, so that a processor can combine them at the join and continue to execute the parent thread. This remote operation is usually made when one of the two threads arrive at the join, but in principle it can happen at any time before the join is executed.

3.3 Work Stealing on Hierarchical Systems

3.3.1 The Class of Work Stealing Algorithms

An algorithm is a work stealing algorithm if processors always follow the rules below:

- Each processor maintains its own pool to store threads it creates.
- A processor currently executing a thread will continue until it arrives at a fork or a join.
- When a processor executing thread x arrives at a fork that creates a new thread y, it puts either x or y in its pool and continues to execute the other.
- When a processor reaches a join, it executes the join if the join is ready to be executed and otherwise tries to pick a thread from its pool to execute.
- If the pool of a processor is empty when the processor tries to take a thread from it, the processor "steals" some threads from the pool of another processor, executes one of them and puts the other in its own pool.

A processor making a steal is a *thief* processor and the processor a thief steals from is a *victim* processor. A steal is a *local steal* if the thief and the victim processor are in the same cluster, and otherwise it is a *remote steal*. A local steal takes time 1 to complete while a remote steal takes at least time r. We assume programs are compute-intensive, but not very data-intensive, such that a remote steal of time $\Theta(r)$ is enough to transfer all necessary information of a thread from one cluster to another.

In order to keep the class of work stealing algorithms as general as possible, we do not put any constraints on which thread a processor takes out from its pool to execute or which threads of which processor a "thief" processor steals.

We will show lower bounds for the class of work stealing algorithms on s-bounded divide-and-conquer programs in Section 3.5. Keeping the class of work stealing algorithms general makes our lower bounds stronger.

3.3.2 Global Work Stealing Algorithm

The classical work stealing algorithm [20] has been proved to perform well in the traditional shared-memory model, both theoretically and empirically. However, little was known about its theoretical time bounds on hierarchical platforms. In this section, we introduce the *global work stealing algorithm*, a variant of the classical work stealing algorithm modified for the hierarchical system model. We will show how to modify the classical work stealing algorithm so that it has a good upper bound on the expected execution time for any fork-join program in the hierarchical system model.

As in the classical work stealing, the pool of a processor to store threads is a doubly-ended queue (deque) in the global work stealing algorithm. When the processor executing thread x arrives at a fork u that creates a new thread y, it pushes either x or y into the bottom of its deque and continues to execute the other. When a processor needs to take a thread out of its deque, it always pops the first one from the bottom of the deque, i.e., the one pushed into the deque at the last fork. When a processor makes a steal, it chooses a processor in the whole system uniformly at random as the victim and steals only the first thread from the top of the deque of the victim.

As we discussed before, a remote steal incurs a remote execution. To be more concrete, we assume a remote steal takes time $2r$ to complete. In the first time r, the thief makes a steal attempt (e.g., by sending a request message to the victim) during which the node to be stolen stays in the victim's deque. At the first time slot of the second interval of time r, the actual steal action is made. If there are multiple steal actions made at the same moment, an arbitrary one will succeed and others fail. No matter if a remote steal succeeds or not, it takes time r for the thief to get the result. Therefore, if the remote steal succeeds, the stolen node is in transition during the second interval of time r, and hence no processor can execute it until it arrives at the thief at the end of the interval.

The algorithm also needs to deal with joins carefully. When a processor in cluster $\mathcal{C}_a$ executes thread x and spawns thread y at a fork, some space in $\mathcal{C}_a$ (e.g., some memory location in $\mathcal{C}_a$) is reserved for the join of x and y. When the processor executing x reaches the join, it first checks the reserved space to figure out if y has already arrived at the join. If so, the processor retrieves the information of y and executes the join. Otherwise, it blocks x and puts x in the reserved space. The processor executing y behaves similarly and the only difference is that the information of y to be written is the result (output) of the computation of y, as y is finished at the join.

As we can see, if x and y are executed by two threads in different clusters, $\mathcal{C}_a$ is remote to at least one of the threads. We say a processor makes a *remote join operation* at the join if $\mathcal{C}_a$ is remote to it. Like a remote steal, a remote join operation takes time $2r$ (it could be up to $4r$ if the processor didn't combine its operations. We keep it simple, as the constant factor doesn't affect our theoretical analysis).

In the classical work stealing algorithm, a steal only takes the thread to be stolen from the victim. Here we make an important change in the global work stealing algorithm: when a processor steals a thread x, it also steals the threads that have been *attached to* x. A thread y can be attached to x only in the following scenario: If a processor executing thread y arrives at the join of thread x and

y, and finds x has not reached the join yet, the processor pops a thread from the bottom of its deque as usual. If the thread popped out happens to be x, the processor *attaches* y *to* x, by attaching to x everything needed for the join of x and y. Since it is an operation within the same cluster, it takes time 1 in our theoretical model. Now that y has been attached to x, the processor executing x can retrieve y locally when it later reaches the join of x and y, without the need to check the reserved space for the join, and hence avoiding the potential risk of making a remote join operation. This attaching scheme resembles the clone optimization [34] in the classic shared-memory setting: the high-level idea of the clone optimization is that, when thread x is completed while thread y has not been stolen yet, y can be modified to a fast version such that when reaching the join of x and y, it can simply resolve the join using the result of x.

Now assume the global work stealing algorithm is *work-first* at forks. That is, a processor at a fork always puts the parent thread into its deque and continues to execute the child thread. Then we can prove that only the child thread can be attached to the parent thread in this case, not vice versa. [2] Therefore, when a thread is completed by a processor and the next thread popped out is the parent thread of the completed thread, the processor can just attach the result of the completed thread to its parent thread. Although the choice at a join doesn't affect the theoretical bounds in the paper, we think attaching only the result of a thread to its parent is easier to implement in practice.

The attaching scheme can largely reduce the number of remote join operations in an execution. Roughly speaking, without this trick, a processor can incur $\Theta(T_\infty)$ remote join operations after a single remote steal of a thread in the worst case. For instance, if all the $\Theta(T_\infty)$ threads spawned by thread x have already been executed by the time x is stolen by a remote processor, that remote processor has to make a remote join operation whenever it reaches the join of x and one of the $\Theta(T_\infty)$ threads. Figure 3.4 shows an example of a single remote steal incurring a sequence of remote joins. As we will show in Lemma 22, the attaching scheme helps us bound the number of remote join operations in an execution, which is critical in the proof of the upper bound on the algorithm's execution time.

3.4 Bounds for Fork-Join Programs

3.4.1 Lower Bound for All Algorithms

We now prove the lower bounds in the following theorem hold for any load balancing algorithm on fork-join programs.

Theorem 19 *Given any T_1 and T_∞, there is a fork-join DAG $\mathcal{G}(T_1, T_\infty)$ that any algorithm takes time $\Omega(\min\{\frac{T_1}{n} + T_\infty, \frac{T_1}{p} + \frac{rT_\infty}{\log(nr)}\})$ to execute on a hierarchical system.*

[2]Lemma 3 in Arora et al.'s paper [7] implies that if the algorithm follows the first-first strategy, the first thread in the bottom of a processor's deque must be an ancestor thread of the thread the processor is executing. This rules out the possibility that the thread popped out from its deque is a child thread of the thread that got just blocked.

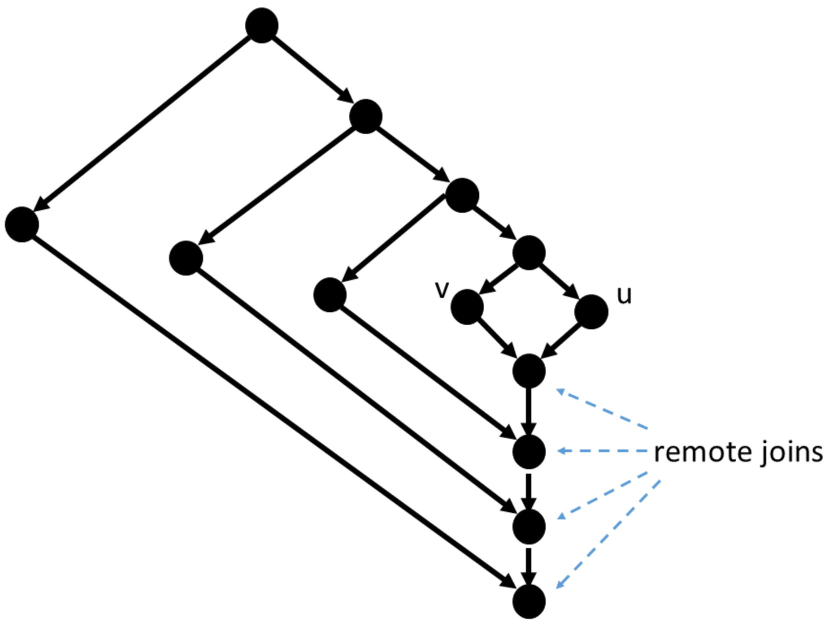


Figure 3.4: A DAG where a single remote steal can incur a sequence of remote join operations. Suppose the processor executing the main thread always execute the child thread first after a fork. At each of the first three joins of the main thread, the processor spawns and quickly completes a child thread, and then gets back to the main thread. It finally arrives at the last fork, spawns a child thread and is about to execute v, while u is pushed into its deque. Now a remote processor steals u and start executing the main thread. If v is executed before u, the remote processor will have to make a remote join operation at each of the four joins in the main thread, in order to retrieve the results of the four child threads. We can imagine if there are a sequence of $\Theta(T_\infty)$ joins, $\Theta(T_\infty)$ has to be made.

Proof. Let us first discuss the case when $\frac{T_1}{T_\infty} \leq \frac{nr}{2\log(nr)-1}$. We will prove the lower bound in this case is $\Omega(\frac{T_1}{n} + T_\infty)$. Without loss of generality and for simplicity, assume $\frac{T_1}{T_\infty} = \frac{2^t+2^{t-1}-2}{2t-1}$, for some integer t. Note that $2^t \leq nr$.

Consider a fork-join DAG $\mathcal{G}(T_1, T_\infty)$ consisting of a sequence of "diamond" structures, as illustrated in Figure 3.5. The first half of each diamond is a complete binary tree of $2^t - 1$ nodes and the second half is the reverse of the first half—nodes keep joining in pairs until it becomes a single node. Obviously, each diamond has $2^t + 2^{t-1} - 2$ nodes and height $h = 2t - 1$. The DAG consists of $\frac{T_\infty}{h}$ such diamonds.

Now consider an arbitrary algorithm that executes this DAG. Suppose the first node of a diamond is executed by a processor in a cluster at time t_0. Since all other nodes of the diamond are descendants of its first node. If there is no remote steal successfully stealing a node of the diamond from that cluster, then the diamond will be executed only by the n processors in the cluster. Thus it takes time $\Omega(\frac{2^t}{n} + h)$ to complete the diamond. If one node of the diamond is stolen by a remote processor, it will take the algorithm at least time r to complete the diamond, as it is the cost to make a remote operation. Since $\frac{2^t}{n} \leq r$ and $h = 2t - 1 < r$, we conclude that the execution time of the diamond

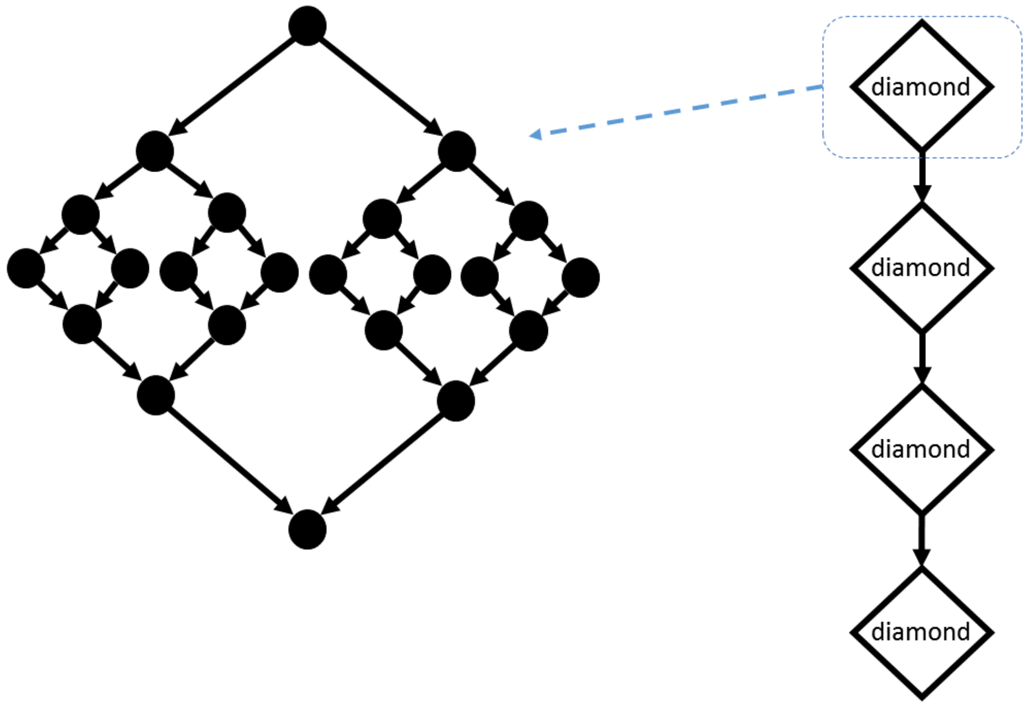


Figure 3.5: The graph on the left presents a diamond structure and the DAG on the right consists of a sequence of four diamonds.

is $\Omega(\frac{2^t}{n} + h)$. Since the $\frac{T_\infty}{h}$ diamonds have to be executed sequentially, the execution time of the entire DAG is $\Omega(\frac{T_\infty}{h} \cdot (\frac{2^t}{n} + h)) = \Omega(\frac{T_1}{n} + T_\infty)$.

Now we discuss the case when $\frac{T_1}{T_\infty} > \frac{nr}{2\log(nr)-1}$. The DAG $\mathcal{G}(T_1, T_\infty)$ we want to construct is shown in Figure 3.6. The first node of the DAG forks into two parallel subgraphs that join at the end of the DAG. The left subgraph is a sequence of $\frac{T_\infty - 2}{h}$ diamonds, each having nr nodes, where h is the height of each diamond and we know $h = \Theta(\log(nr))$ (again, without loss of generality and for simplicity, assume $\frac{T_\infty-2}{h}$ is an integer and nr nodes are just enough to construct a diamond). The right subgraph consists of the rest $(T_1 - 2 - \frac{nr(T_\infty-2)}{h})$ nodes in an arbitrary form with only one requirement that its critical path length is at most $T_\infty - 2$. As we analyzed before, the execution time of the left subgraph is $\Omega(\frac{T_\infty-2}{h} \cdot (\frac{nr}{n} + h)) = \Omega(\frac{rT_\infty}{\log(nr)})$. On the other hand, since there are p processors in the system, the execution time of the entire DAG is $\Omega(\frac{T_1}{p})$. Hence the lower bound on the execution time is $\Omega(\frac{T_1}{p} + \frac{rT_\infty}{\log(nr)})$.

Now combining the lower bounds in the two cases we have discussed above, we complete the proof. □

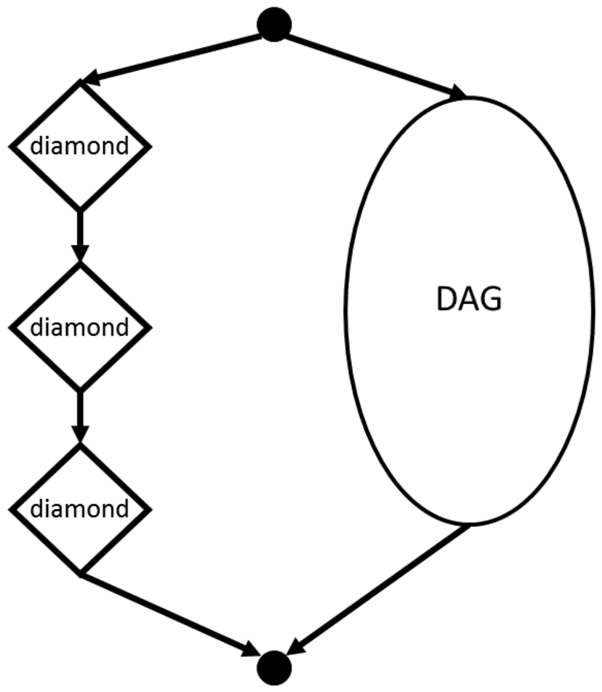


Figure 3.6: A DAG $\mathcal{G}(T_1, T_\infty)$, where the subgraph on the left branch of the source node is a sequence of diamonds, and the subgraph on the right branch of the source node is an arbitrary DAG.

3.4.2 Upper bounds of Two Work Stealing Algorithms

In this section, we will show two work stealing algorithms, *the local work stealing algorithm* and the global work stealing algorithm, achieve good uppers for different ratios of $\frac{T_1}{T_\infty}$.

In the local work stealing algorithm, a processor steals uniformly at random from a processor within the same cluster and hence all steals are local. Since in our theoretical model, a DAG is induced from a single source node held by a processor in a cluster at the start of an execution, the DAG will be executed only by the n processors in the cluster in the local work stealing algorithm. Therefore, the local work stealing algorithm is essentially the classical work stealing running in a single cluster. (Although in practice, different programs that have synchronization points with each other can initially be assigned to different clusters, and those programs together may be considered as a DAG.) Thus, the upper bound of the classical work stealing algorithm in the shared memory model by Arora et al. [7] immediately gives the following bound.

Claim 20 *The local work stealing algorithm executes any DAG $\mathcal{G}(T_1, T_\infty)$ in expected time $O(\frac{T_1}{n} + T_\infty)$ on a hierarchical system.*

Comparing Claim 20 with Theorem 19, we know the local work stealing algorithm is optimal when $\frac{T_1}{T_\infty} \leq \frac{nr}{\log(nr)}$.

Since most of the steals in the global work stealing algorithm are remote, each taking time $2r$, one may think the algorithm should be quite inefficient for many programs. Surprisingly, the theorem below shows the global work stealing algorithm actually achieves a very good upper bound, when the the workload of a program is heavy (i.e., $\frac{T_1}{T_\infty}$ is large).

Theorem 21 *The global work stealing algorithm executes any DAG $\mathcal{G}(T_1, T_\infty)$ in expected time $O(\frac{T_1}{p} + rT_\infty)$ on a hierarchical system.*

It is easy to see that, when $\frac{nr}{\log(nr)} \leq \frac{T_1}{T_\infty} \leq pr$, the global work stealing algorithm is optimal within a factor of $\Theta(\log(nr))$, and when $\frac{T_1}{T_\infty} \leq pr$, the global work stealing algorithm is optimal. Intuitively, doing remote steals frequently can quickly balance the workload across the system. Also, when the workload is heavy, the cost of remote steals will be negligible compared to the cost of executing the program itself.

The rest of the section is the proof of Theorem 21. We start with the following lemma that bounds the number of remote joins in an execution.

Lemma 22 *In any execution of a fork-join program by the global work stealing algorithm on a hierarchical system, the number of remote join operations is at most twice the number of successful steals.*

Proof. Suppose a thread x creates a new thread y at a fork u and the two thread will join at node v. Without loss of generality, assume the processor p_0 currently executing x pushes x into the bottom of its deque and then starts executing y. Since it is a fork-join program, we know that the subgraph induced from y is a fork-join DAG, denoted by $\mathcal{G}_y$, whose source and sink nodes are u and v respectively. Assume x is not stolen by any other processor. If p_0 needs to push nodes at forks in $\mathcal{G}_y$ into its deque, the nodes will be put below x in its deque. Since other processors steals nodes from the top of the deque, no nodes in $\mathcal{G}_y$ can be stolen and hence p_0 will execute all nodes in $\mathcal{G}_y$ and finally completes y. Now p_0 will immediately pop x from its deque and attach the result of y to x. Therefore, if x is not stolen, neither of the two join operations for v is remote. In other words, the two join operations for v can be remote only if x is stolen. Therefore the number of remote join operations is at most twice the number of successful steals. $\square$

Lemma 23 *In any execution of $\mathcal{G}(T_1, T_\infty)$ by the global work stealing algorithm on a hierarchical system, the expected number of steals is $O(pT_\infty)$.*

Proof. (Sketch) The proof is a variant of the well-known proof (of Corollary 4 and Lemmas 6–8) in Arora et al.'s paper [7]. The main idea of the proof in Arora et al.'s paper [7] is to analyze the decrease of the potential Φ which is $3^{\Theta(2T_\infty)}$ at the beginning of an execution. The decrease of Φ can be caused by making steals and executing nodes. The proof by Arora et al. shows that Φ decreases by a constant factor with a constant probability after a period in which p steals are made. Therefore, Φ becomes 0, which indicates the end of the execution, after $O(pT_\infty)$ steals in expectation.

We only have to make one change to Arora et al.'s proof, in order to apply it to the global work stealing algorithm on a hierarchical system: we show that Φ decreases by a constant factor with a constant probability after a period in which $2p$ remote steals are made (instead of p steals). The idea behind it is that each remote join operation takes time $2r$ on a hierarchical system and hence the decrease of Φ caused by a remote join operation takes affects in time $2r$. Since each remote steal also takes time $2r$, we know that after $2p$ remote steals have been made, all the remote joins that occurred during the period in which the first p of the $2p$ remote steals were done must have been completed. The rest of the proof is identical to those of Corollary 4 and Lemmas 6–8 in Arora et al.'s paper [7] and hence we omit it due to space limits.

We complete the proof by observing that the total number of steals has the same bound, as in expectation the number of remote steals is $\frac{k-1}{k}$ of the total number of steals. □

Now we are ready to prove Theorem 21.

Proof of Theorem 21. Processors in a work stealing algorithm only have three kinds of actions—executing a node, making a steal, and doing a remote join operation. Suppose an execution finishes in time T. Thus, we know that $pT \leq T_1 + 2r \cdot (N_s + N_j)$ and hence $T \leq \frac{T_1}{p} + \frac{2r(N_s+N_j)}{p}$, where N_s is the number of steals and N_j is the number of remote join operations. By Lemmas 22 and 23, $N_s = O(pT_\infty)$ in expectation and $N_j \leq N_s$. Therefore $T = O(\frac{T_1}{p} + rT_\infty)$ in expectation. □

3.5 Bounds for s-Bounded Programs

3.5.1 Lower Bound for Work Stealing Algorithms

We now discuss the bounds for work stealing on s-bounded programs which we think are the programs we encounter most in practice.

Theorem 24 below shows that no work stealing algorithm can beat the lower bound in Theorem 19 even for s-bounded divide-and-conquer algorithms, a sublass of s-bounded fork-join programs.

Theorem 24 *Given any $T_1 \geq 8nr$ and T_∞, and any positive integer $s = O(r)$, there is a divide-and-conquer DAG $\mathcal{G}(T_1, T_\infty, s)$ that any work stealing algorithm takes expected time $\Omega(\min\{\frac{T_1}{n} + T_\infty, \frac{T_1}{p} + \frac{rT_\infty}{\log(\frac{nr}{s})}\})$ to execute on a hierarchical system.*

Proof. When $T_1 = O(nT_\infty)$, $\frac{T_1}{n} + T_\infty = O(T_\infty)$. Therefore the bound $\Omega(\frac{T_1}{n} + T_\infty)$ holds in this case, as $\Omega(T_\infty)$ is a lower bound for any DAG whose critical path length is T_∞. The rest of the proof will focus on the case when $T_1 > nT_\infty$.

For simplicity, we will hide some annoying constants in the proof and use asymptotical notations (usually the Θ notation) instead, as the lower bounds we want to prove won't be affected this way and our proof is essentially similar to that of Theorem 19.

The main idea is to construct a series of "hexagon" structures recursively. As shown in 3.7, the first part of a hexagon is a complete binary tree of $\max\{\frac{4nr}{s}, 4n\}$ leaves and of height $\Theta(\log(\frac{nr}{s}))$. For simplicity, we assume $s \leq r$ and hence there are $4nr$ leaves. (When $s > r$, the proof is almost

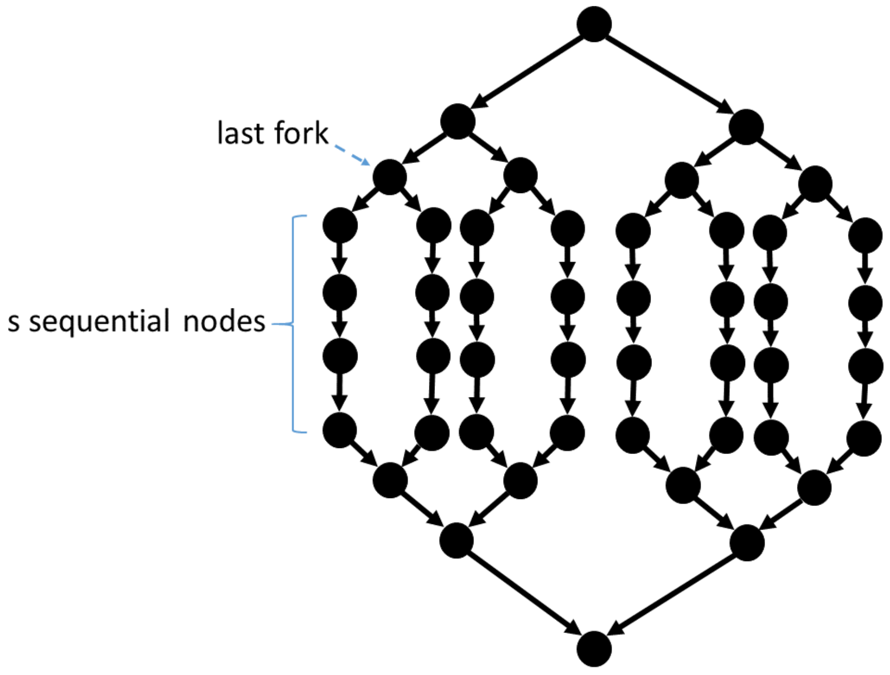


Figure 3.7: A hexagon where each thread has $s = 4$ sequential nodes between the last level of forks and the first level of joins.

identical and hence we omit it in the paper.) All the $\frac{4nr}{s}$ leaves each spawn a sequence of s nodes, representing $\frac{4nr}{s}$ threads each having s nodes to execute after their last forks. Finally, the threads join in pairs at recursively and eventually join into a single node.

The series of hexagons we want to construct is illustrated in Figure 3.8. We choose one of the $\frac{4nr}{s}$ threads in the first hexagon uniformly at random, and replace its s nodes between the last fork and the first join with another hexagon. recursively, we replace the s sequential nodes of a random thread in the ith hexagon with the $(i+1)$th hexagon until we reach the last hexagon we want to create. If there are t hexagons, it is not hard to see that the whole structure has $S_t = \Theta(tnr)$ nodes and height $H_t = \Theta(t \log(\frac{nr}{s}))$, for $t \geq \frac{s}{\log(\frac{nr}{s})}$.

Now consider the case when $8nT_\infty < T_1 \leq \frac{nrT_\infty}{\log(\frac{nr}{s})}$. Suppose $T_1 = \Theta(\frac{dnT_\infty}{\log(\frac{nr}{s})})$, for some integer d such that $\log(\frac{nr}{s}) < d < r$. Let t be the largest integer such that $S_t < T_1$, where, as we defined before, S_t is the number of nodes in a series of t hexagons. The DAG we construct is a series of t hexagons, with the last node of the first hexagon followed by an arbitrary DAG $\mathcal{G}'(T_1 - S_t, T_\infty - H_k)$ in order to make the entire DAG a $\mathcal{G}(T_1, T_\infty, s)$. It is not hard to prove that $S_t = \Theta(T_1)$ and $t = \Theta(\frac{dT_\infty}{r\log(\frac{nr}{s})})$, when $T_1 > 8nr$ (so that the nodes are enough to construct a DAG with as least one hexagon).

We will now prove that the expected execution time of the t hexagons by any work stealing algorithm is $\Omega(\frac{dT_\infty}{\log(\frac{nr}{s})})$ and hence the bound $\Omega(\frac{T_1}{n} + T_\infty) = \Omega(\frac{dT_\infty}{\log(\frac{nr}{s})})$ holds in this case. It suffices to prove that, after reaching the first node of a hexagon, it takes any algorithm expected time $\Omega(r)$ to reach the next hexagon.

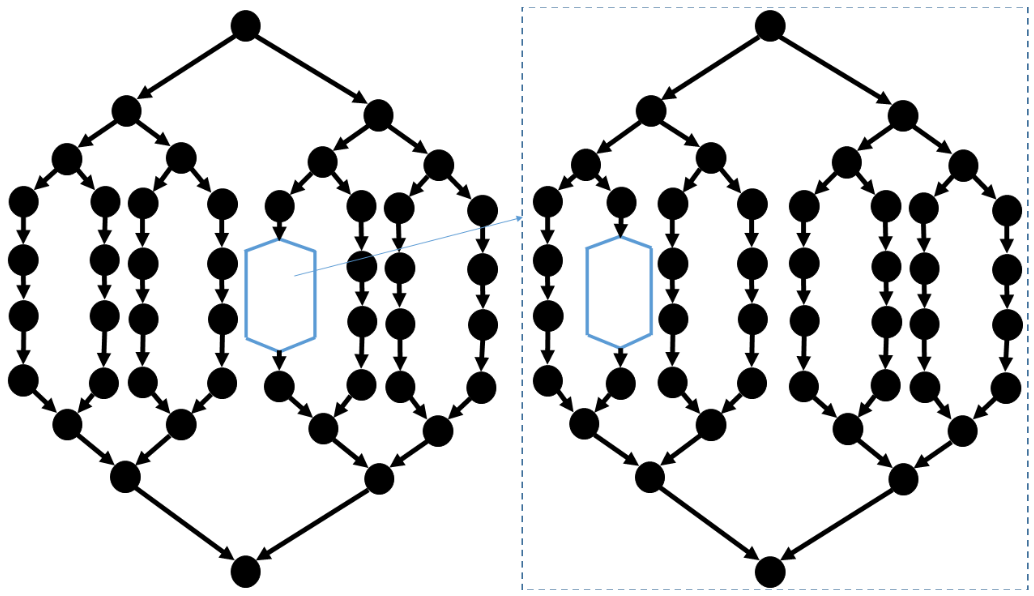


Figure 3.8: The first two hexagons in a series of hexagons. The s sequential nodes of a randomly chosen thread in the first hexagon (on the left-hand side) is replaced by the second hexagon.

Suppose the first node of a hexagon is about to be executed by a processor in a cluster. If a remote processor steals a node that is an ancestor of the next hexagon, it is obvious that it takes at least time r for the node to get to the remote processor, and hence time $\Omega(r)$ to reach the next hexagon in this case. Since there are $\frac{4nr}{s}$ threads in the hexagon and those threads are identical before the last layer of forks, we can conclude that if the first node of the next hexagon has not been reached and remote processors have stolen nodes that will spawn more than $\frac{2nr}{s}$ threads of the current hexagon, then with probability at least $\frac{1}{2}$ the thread spawning the next hexagon is in those threads. Hence the lower bound of expected time $\Omega(r)$ holds in this case.

On the other hand, suppose remote processors only steal nodes that will spawn no more than $\frac{2nr}{s}$ threads and the one spawning the next hexagon remains in the cluster. Now the n processors in the cluster will have to find out the thread containing the next hexagon out of more than $\frac{2nr}{s}$ seemingly identical threads. Note that if a processor in a work stealing algorithm reaches the first node of a thread after the last fork, it won't do anything else until it finishes all the s nodes. Therefore, with probability at least $\frac{1}{2}$ the first $\frac{nr}{s}$ threads whose nodes after the last layer of forks are executed do not include the one spawning the next hexagon and the execution time is already $\Omega(\frac{nr\cdot s}{s\cdot n}) = \Omega(r)$. This completes the proof for the case when $8nT_\infty < T_1 \leq \frac{nrT_\infty}{\log(\frac{nr}{s})}$.

Now we prove the bound $\Omega(\frac{T_1}{p} + \frac{rT_\infty}{\log(\frac{nr}{s})})$ for the case when $T_1 > \frac{nrT_\infty}{\log(\frac{nr}{s})}$. Since $\Omega(\frac{T_1}{p})$ always holds for any system of p processors, we will prove $\Omega(\frac{rT_\infty}{\log(\frac{nr}{s})})$. The proof is similar to that of Theorem 19 and hence we will keep it brief. The DAG $\mathcal{G}(T_1, T_\infty, s)$ we want to construct is similar to the one in Figure 3.6: a single node forking into two subgraphs that eventually join. The left subgraph is a series of $\Theta(\frac{T_\infty}{\log(\frac{nr}{s})})$ hexagons that we explained above and the right subgraph contains the rest of nodes in an arbitrary form. As we proved before, it takes any work stealing algorithm expected time $\Omega(r \cdot \frac{T_\infty}{\log(\frac{nr}{s})})$ to execute the left subgraph, which completes the proof. □

3.5.2 Unbalanced Work Stealing and Its Upper Bound

We say a work stealing algorithm is an *unbalanced work stealing algorithm*, if in any execution, 1) each processor pushes and pops threads through the bottom of its deque, 2) each local steal of a processor chooses a victim processor uniformly at random from the same cluster, 3) each remote steal of a processor chooses a victim processor uniformly at random from other clusters, 4) each steal takes only the first thread from the top of the victim's deque, 5) after a processor made a remote steal, its next steal is a local steal with probability at least c, for a constant c, and 6) $N_{rem} \leq N_{loc} \leq \lambda \cdot N_{rem}$, where $\lambda = \Theta(r)$, N_{rem} is the number of remote steals and N_{loc} is the number of local steals in the execution.

The first three rules are standard in most work stealing algorithms in the literature. Rule 4 is also the strategy many existing algorithm uses. However in other algorithms, a thief processor can steal multiple threads (e.g., half of the threads) from a deque. Rule 5 implies that after making a remote steal, a processor is likely to make a local steal. Rule 6 means intuitively that the algorithm has a bias towards local steals while still making enough remote steals.

The most common heuristic we find in the existing work stealing algorithms for hierarchical systems is to have each processor make more local steals than remote steals, hoping that this can reduce the number of costly remote operations while most processors can have enough work to do by only balancing workloads locally most of the time [62].

Algorithms using this heuristic usually behave like unbalanced work stealing algorithms, at least when running on unbalanced programs in which workloads cannot keep evenly distributed among clusters for long without remote steals. Since unbalanced programs are usually the ones on which algorithms performance worst in practice, we believe we can measure the performance bounds of many work stealing algorithms using this heuristic, by proving the bounds of the class of unbalanced work stealing algorithms.

When the workload is light, the local work stealing algorithm has an optimal upper bound, as we showed before. Since algorithms with this heuristic are in favor of local steals, they should work well in this case. Theorem 25 indicates that algorithms using this heuristic are likely to have good performance guarantees for $\Omega(r)$-bounded fork-join programs, when the workload is heavy, i.e., $\frac{T_1}{T_\infty}$ is big, and each thread contains a relatively large amount of work.

Theorem 25 *An unbalanced work stealing algorithm executes any s-bounded DAG $\mathcal{G}(T_1, T_\infty)$ in expected time $O(\frac{T_1}{p} + rT_\infty)$ on a hierarchical system, for any $s = \Omega(r)$.*

The proof of Theorem 25 is based on the following lemma.

Lemma 26 *In any execution of a DAG $\mathcal{G}(T_1, T_\infty, s)$ by any work stealing algorithm on a hierarchical system, the number of joins is no more than $\frac{T_1}{s}$.*

Proof. The proof is simple. Since each thread in an s-bounded DAG $\mathcal{G}(T_1, T_\infty)$ has at least s nodes and there are in total T_1 nodes in the DAG, the number of threads is at most $\frac{T_1}{s}$. Since a join in a fork-join DAG is the end of a thread, the number of joins is bounded by the number of threads in the DAG and hence is at most $\frac{T_1}{s}$. □

Proof of Theorem 25. By the definition of unbalanced work stealing algorithms, after a processor made a remote steal, its next steal is a local steal with probability at least c, for some constant c. Suppose a processor makes m remote steals in an execution. We call each odd-numbered one of these m remote steals and the steal of the processor after it (which is a local steal with probability at least c) a *steal couple*. Thus, the processor has $\lceil m/2 \rceil$ steal couples and they don't overlap. It is easy to see that by making a steal couple, the processor steals each other processor in the system with probability at least $\min\{\frac{1}{n(k-1)}, \frac{1}{c(n-1)}\} > \frac{1}{cp}$. Therefore, by the same proof of Lemma 23, we can prove that the expected number of steal couples by all processors in an execution is $O(pT_\infty)$. Hence, the expected number of remote steals in an execution is also $O(pT_\infty)$.

The rest of the proof is similar to that of Theorem 21. The execution time of an unbalanced work stealing algorithm is $T \leq \frac{T_1}{p} + \frac{2r(N_{rem}+N_j)+N_{loc}}{p}$, where N_{rem} is the number of remote steals, N_{loc} is the number of local steals, and N_j is the number of remote join operations. Since $N_{rem} = O(pT_\infty)$ in expectation, $N_{loc} = O(rN_{rem})$ by definition, and $N_j = O(\frac{T_1}{r})$ by Lemma 26, we have $T = O(\frac{T_1}{p} + rT_\infty)$ in expectation. □

As a byproduct, the following theorem shows that The global work stealing algorithm, even without the attaching scheme, has the same upper bound for $\Omega(r)$-bounded fork-join programs.

Theorem 27 *The global work stealing algorithm, even without the attaching scheme, executes any DAG $\mathcal{G}(T_1, T_\infty, s)$ in expected time $O(\frac{T_1}{p} + rT_\infty)$ on a hierarchical system, for any $s = \Omega(r)$.*

Proof. The proof is similar to that of Theorem 21. Lemma 26 indicates any work stealing algorithm incurs only $O(\frac{T_1}{r})$ remote join operations in an execution of a DAG $\mathcal{G}(T_1, T_\infty, s)$, when $s = \Omega(r)$. Also note that Lemma 23 still holds for s-bounded DAGs. The execution time of the global work stealing algorithm without the attaching scheme is $T \leq \frac{T_1}{p} + \frac{2r(N_s+N_j)}{p}$, where N_s is the number of steals and N_j is the number of remote join operations. Since $N_s = O(pT_\infty)$ in expectation and $N_j = O(\frac{T_1}{r})$, we have $T = O(\frac{T_1}{p} + rT_\infty)$ in expectation. □

Chapter 4

Work Stealing for Linearizable Futures

This chapter presents our work on linearizable futures, a new type of futures invented recently [53] to handle functions with side effects, especially for method calls to long-lived shared data structures. we propose a new program model, called the *linearizable-futures model*, that supports both futures without side-effects, which we call *normal futures*, and futures that exist for their side-effects, which we call *linearizable futures*. We use this model to propose a novel work-stealing scheduler that facilitates the kind of combining and elimination optimizations supported by linearizable futures.

The rest of the chapter is organized as follows. Section 4.1 shows the related work. In Section 4.2, we explain our new program model, linearizable-futures model, in detail. In Section 4.3, we show how to modify work stealing for linearizable-futures model, in order to make good use of combining and elimination optimizations. Finally we prove in Section 4.4 that the modified work stealing, combined with combining and elimination, achieves very good performance bounds with respect to its execution times on programs in linearizable-futures model.

4.1 Related Work

Futures were first proposed by Halstead [41, 42] and have been well studied since then (e.g., [55, 8, 58, 31, 16, 77, 32]), sometimes under different names. They are a flexible way to structure parallel programs, as the future-parallel model is a generalization of the widely used fork-join model (also called nested-parallel model) [13, 15, 17].

Futures are well suited to dynamic load-balancing techniques such as the popular work stealing scheduler [20]. Arora et al. [7] proved that (parsimonious) work stealing achieves the asymptotically best speedup for the parallel execution of a program in the future-parallel model. If a program in the future-parallel model is written in a natural, structured way [47], its parallel execution using work stealing also guarantees good cache locality, much better than a worst-case unstructured program [1, 73].

Kogan and Herlihy [53] recently proposed futures for side-effects, intended to facilitate batching optimizations such as combining [39, 37, 46] and elimination [45, 63, 72]. An operation on a shared data structure immediately returns a future, and a later touch will retrieve the return value of the operation, as soon as that value is ready. Since operations on shared objects have side-effects, it is necessary to specify when these side-effects might be observed. Three alternative correctness conditions were proposed: *strong*, *medium*, and *weak futures linearizability.* These conditions should be thought of as extensions to linearizability [49], a widely used correctness condition in distributed and parallel computing.

Here, we restrict out attention to medium futures linearizability, arguably the most useful of the futures-based extensions to linearizability. For brevity, futures satisfying this condition are called *linearizable futures* here.

4.2 Linearizable-Futures Model

4.2.1 Normal Futures and Linearizable Futures

As noted, futures [41, 42] are an attractive way to structure parallel computation. When a thread creates an expression (usually a method call) with a keyword *future*, a new thread, called a *future thread* or a *future* for short, is spawned to compute that expression in parallel with the thread that created it. When a thread needs the result of that expression, it applies a *touch* operation to that future. If the result is ready, it is returned to the touching thread, and otherwise the touching thread blocks until the result becomes ready (see Figure 4.1 for an example). A future thread itself can even create new futures.

```
Future x = Future fibonacci(3); // fibonacci(3) and fibonacci(5) are
Future y = Future fibonacci(5); // method calls created as futures

            ⋮

Int a = x.touch() + y.touch();
```

Figure 4.1: Pseudocode for computing the sum of the third and the fifth Fibonacci numbers using futures.

In prior models, futures are deterministic in a program (with a given input): the sequence of a future thread's instructions are fixed in all executions, no matter how those instructions are interleaved with other threads. Once we introduce threads with asynchronous side-effects, we can no longer guarantee determinism. Instead, we allow future threads to be *regular*: in any execution of a program, (1) the number of a regular thread t's instructions is fixed, (2) the futures and touches

t creates, as well as their positions in t, are fixed,[1] and (3) the instructions between two successive touch operations in t take effect as if they were executed atomically at the moment the first of the two touches is executed.[2] We call futures that are regular *normal futures*, to distinguish them from *linearizable futures* introduced next.

Kogan and Herlihy [53] recently proposed a new way of using futures to encapsulate operations on shared mutable data structures. When an operation is applied to a shared data structure, it immediately returns a future. Touching that future later returns the operation's result, along with implicit confirmation that the operation has taken effect. We call futures for operations on shared objects *linearizable futures.* A linearizable future for an operation on a shared object o is also called a linearizable future on o, for short.

In the presence of concurrency, it is important to specify when a future-returing operation can take effect. Three alternative correctness conditions have been proposed: strong, medium, and weak futures linearizability [53]. In this paper, we focus on *medium futures linearizability*, which appears to be the most useful.

Medium futures linearizability requires that (1) the operation associated with a linearizable future should appear to take effect at some instant between when that future is created, and when it is touched, and (2) linearizable futures created by the same thread on the same object should take effect in the same order as their creation operations. Informally, these conditions mean that (1) the creation of a linearizable future and its matching touch fill the roles of operation invocation and response in the classic linearizability condition, and (2) the order of operations called by a thread on the same shared object is preserved.

4.2.2 The Model

In this paper, we propose a new parallel program model, called *linearizable-futures model*, that exploits both normal futures and linearizable futures in a structured way. Suppose there are multiple main threads (which represent multiple independent programs in practice) in a program, each having its own, side-effect free computation. The main threads also need to communicate with each other by accessing shared objects in the shared memory. As we can imagine, an efficient way to run this program is to have each main thread spawn "worker threads" to execute its local, computationally heavy work in parallel, and also have each main thread create linearizable futures to apply operations to shared objects in order to communicate with other main threads. Our linearizable-futures model captures this scenario.

More specifically, a program in linearizable-futures model consists of some independent *main threads* and their descendant threads which are normal futures and linearizable futures. The main

[1] In fact, the number of instructions of a thread is allowed to vary within a constant factor, and the positions of futures and touches can move within a constant number of instructions, as long as their order is preserved. We ignore these variations, as they are negligible in our theoretical performance analysis, and do not seem useful in practice.

[2] Intuitively, the instructions between two successive touches are required to be side-effect free. We define it this way, because in our new program model, introduced later, only the touch operations in a regular thread can return data from shared objects, causing nondeterminism in the thread.

threads and the normal futures are *regular*. A main thread can spawn and touch both normal futures and its own linearizable futures. A normal future thread, which is spawned directly or indirectly by a main thread, can only create and touch normal futures, but not linearizable futures, as linearizable futures are designed for operations on shared objects by main threads. A linearizable future thread can neither create nor touch any type of futures, since in practice an operation on a shared object is executed by a single thread. We also require each future to be touched only once (note that the thread touching a normal future doesn't have to be the one that created that future). This requirement is beneficial in practice for reasons such as that it makes the implementations of future/touches simpler and have lower overheads [16], and that the requirement, together with another practical constraint proposed in the well-structured future-parallel model in [47], guarantees good cache locality bounds for parallel executions by work stealing scheduler.

We believe that writing a parallel program in linearizable-futures model is easy and natural: When we want to execute some side-effect free computation in a thread in parallel, or when we want to split a thread into two or more to run different parts of a program later, we create a normal future in that thread, and we may have the newly spawned normal future thread create more normal futures further, if necessary; When a main thread needs to exchange information with other main threads, we have it create a linearizable future to apply an operation to a shared object, which is the standard way of communication in shared-memory systems.

Unlike the previous future-parallel models, our model supports nondeterminism by the use of linearizable futures, with a clear correctness condition—medium futures linearizability. Although the requirement that main threads and normal futures must be regular limits the degree of non-determinism, in return it avoids a program to be notoriously hard to reason about and debug, by forcing nondeterminism to be caused only by single-purpose linearizable future threads and keeping the structure of the program unchanged in all parallel executions.

4.2.3 Computation DAG

For theoretical analysis, an execution of a program is modeled as a *directed acyclic graph* (DAG). A node in a DAG represents an instruction and a directed edge (u, v) represents the dependency constraint that v must be executed after u. Therefore, a node is able to execute when all its parent nodes have been executed. A node that creates a future is called a *fork* and a node that touches a future is called a *touch*. The most common edges in a DAG are *continuation edges*, which point from one node to the next in the same thread. Thus, a *thread* is a maximal chain of nodes connected by continuation edges. There are three other types of edges (Figure 4.2 shows a DAG consisting of edges of all these types):

- *future edges*, which point from a fork u to the first node of the future thread spawned by u.
- *touch edges*, which point from the last node of a future thread f to a touch v in another thread that touches the future computed by f.

- *order edges*, which point from the last node of a linearizable future f_1 on object o created by a thread t to the first node of the next linearizable future f_2 on o created by t, indicating that f_2 cannot be executed until f_1 has be finished, according to medium futures linearizability.

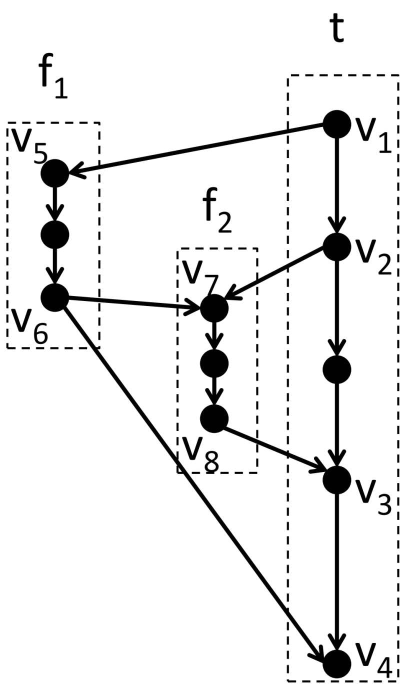


Figure 4.2: A main thread t spawns two linearizable futures f_1 and f_2, both on object o, at forks v_1 and v_2 respectively. v_3 and v_4 are touches of f_1 and f_2 respectively. By definition, (v_1, v_5) and (v_2, v_7) are future edges. (v_6, v_4) and (v_8, v_3) are touch edges. (v_6, v_7) is an order edge. All the other edges are continuation edges.

A DAG consists of some main threads that don't have dependency constraints on each other, and some normal futures and linearizable futures spawned directly or indirectly by them. Since each node is a single instruction, it cannot spawn multiple futures or touch multiple futures. Combining this with the fact that each future can be touched only once, we can conclude that each node has in-degree and out-degree either 1 or 2, except that the first and the last nodes of each main thread have in-degree 0 and out-degree 0, respectively. A *critical path* of a DAG is a longest directed path in the DAG and the length (i.e., the number of nodes) of a critical path is sometimes called the *computation span* of the DAG. In the next subsection, we will show how to generalize the DAG model to represent programs that support combining and elimination optimizations.

4.2.4 Combining and Elimination

Kogan and Herlihy [53] shows that if shared object operations are created as futures, some powerful optimizations, such as combining and elimination, can be applied to a program. Since a linearizable future doesn't have to be executed immediately after its creation, a clever scheduler can delay its execution until it can be done efficiently. For instance, consider a shared queue implemented as a linked list and k enqueue operations created as k linearizable futures by a thread. If a thread can execute those k enqueue operations together, it can locally make them a linked list in the same order as they were created, and then append them to the linked list of the shared queue by only one CAS operation. Since local operations are usually much faster than operations on a shared object, this *combining* optimization will largely boost the performance of the execution, compared to simply enqueuing k elements one by one. Similarly, if a push operation and a pop operation on the same shared stack object can be combined locally, they can be canceled out by an *elimination* optimization, without even accessing the shared stack.

As the main results of the paper, we will show in Section 4.4 that work stealing scheduler, with some modifications, can make good use of combining and elimination optimizations in our model. As Kogan and Herlihy did in [49], we only consider combining and eliminating linearizable futures on the same object created by the same thread. Modeling and analyzing the performance of combining and eliminating linearizable futures created by different threads is out of the scope of the paper, and we consider it an interesting open question for future work.

Since we focus on the analysis of "local" combining and elimination, we make a simplification to our model to assume that the length of the thread that *groups* some linearizable futures to execute together (with possible combining and elimination) is fixed, regardless of the execution of any other part of the program. For instance, let f_0, f_1, f_2 and f_3 be the linearizable futures on object o created by the same thread t. Grouping f_1 and f_2 to execute together will replace these two future threads by a new thread $g(f_1 \cup f_2)$ in the DAG, as illustrated in Figure 4.3, and the length of $g(f_1 \cup f_2)$ is fixed in all executions where f_1 and f_2 are grouped with no other futures. Since we cannot group f_1 and f_2 until f_2 is created, $g(f_1 \cup f_2)$ is spawned at the fork of f_2. The touch of $g(f_1 \cup f_2)$ is the same as the earlier one of the touches of f_1 and f_2, which is the touch of f_2 in this case. Because of medium futures linearizability, there is an order edges from the last node of f_0 to the first node of $g(f_1 \cup f_2)$, and an order edges from the last node of $g(f_1 \cup f_2)$ to the first node of f_3. Note that only consecutive linearizable futures can be grouped, according to medium futures linearizability (e.g., if f_0 and f_2 are grouped, then executing f_1 either before or after $g(f_0 \cup f_2)$ violates medium futures linearizability).

Formally, let $f_0, f_1, ..., f_n$ denote all the linearizable futures on object o created by thread t in a program, where f_i is created before f_{i+1} for all $0 \leq i \leq n-1$. If a thread, denoted by $g(f_i \cup f_{i+1} \cup ... \cup f_j)$, takes $f_i, f_{i+1}, ..., f_j$ to execute together in an execution, for some $0 \leq i \leq j \leq n$, we say $g(f_i \cup f_{i+1} \cup ... \cup f_j)$ is a *future group* (or a *group* for short) in that execution. In the DAG of that execution, future threads $f_i, f_{i+1}, ..., f_j$ are replaced by thread $g(f_i \cup f_{i+1} \cup ... \cup f_j)$, where $g(f_i \cup f_{i+1} \cup ... \cup f_j)$ is created at the fork that creates f_j and touched by the earliest one among the touches of $f_i, f_{i+1}, ..., f_j$. If $i > 0$, there is an order edge from the last node of the group f_{i-1} is in, to the first node of $g(f_i \cup f_{i+1} \cup ... \cup f_j)$. If $i < n$, there is an order edge from the last node of $g(f_i \cup f_{i+1} \cup ... \cup f_j)$ to the first node of the group f_{j+1} is in.

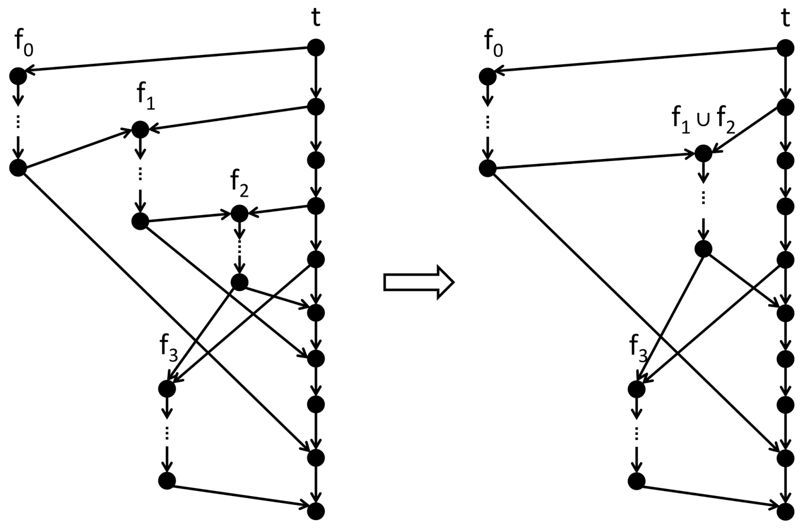


Figure 4.3: The DAG on the left shows an "original" execution of the program, where each linearizable future f_i is executed solely, for any $0 \leq i \leq 3$. The DAG on the right shows the DAG of an execution, where f_1 and f_2 are grouped.

If a linearizable future f_k is executed solely in an execution, we still consider it as a future group $g(f_k)$, in order to keep notations consistent. Thus, all linearizable futures are replaced by future groups in the DAG of an execution. When we say the fork (resp. the touch) of a linearizable future f in a DAG, we refer to the node where f is created (resp. touched) in the original program, although that node is not necessarily the fork (resp. the touch) of a future group in the DAG. When we say the DAG of a program, we refer to the DAG of an execution of the program where each linearizable future is executed solely as a group.

Let $T(g)$ denote the number of nodes of group g, i.e., the length or the execution time of g. If both combining and elimination can be applied to the program whenever possible, we assume $T(g(f_i \cup f_{i+1} \cup ... \cup f_j))$ is fixed in any execution of the program and satisfies the *inequalities of combining and elimination*:

$$0 \leq T(g(f_i \cup f_{i+1} \cup ... \cup f_j)) \leq T(g_1) + T(g_2) + ... + T(g_k)$$

where $g_1, g_2, .., g_k$ are a *partition* of $f_i, f_{i+1}, ..., f_j$, i.e., future groups that execute disjoint subsets of f_i, $f_{i+1}, ..., f_j$ and collectively execute all $f_i, f_{i+1}, ..., f_j$, such that if f_a is in g_c and f_b is in g_d for some $i \leq a < b \leq j$, then $1 \leq c \leq d \leq k$.

The two bounds in the inequalities of combining and elimination show that (1) in the best case, where $f_i, f_{i+1}, ..., f_j$ can be entirely canceled out, no work is needed, (2) executing $f_i, f_{i+1}, ..., f_j$ together in a single group won't be less efficient than splitting them into subgroups to execute one by one (because a group can always simulate the execution of its subgroups if that is the most efficient way), and (3) in the worst case $T(g(f_i \cup f_{i+1} \cup ... \cup f_j)) = T(g(f_i)) + T(g(f_{i+1})) + ... + T(g(f_j))$, where no optimization can be applied to those futures, then $g(f_i \cup f_{i+1} \cup ... \cup f_j)$ has to execute those linearizable futures one by one. Note that the inequalities of combining and elimination is a very weak assumption, allowing the performance of a future group be to anything between the two extreme cases.

If only combining is applied to the program, then we assume $T(f_i \cup f_{i+1} \cup ... \cup f_{i+k})$ is fixed in any execution and satisfies the *inequalities of combining*:

$$T(g^*) \leq T(g(f_i \cup f_{i+1} \cup ... \cup f_j)) \leq T(g_1) + T(g_2) + ... + T(g_k)$$

where g^* is a *subgroup* of $g(f_i \cup f_{i+1} \cup ... \cup f_j)$, i.e., a future group $g(f_a \cup f_{a+1} \cup ... \cup f_b)$, for some $i \leq a \leq b \leq j$.

The inequalities of combining capture the fact that if we cannot cancel out operations, combining a group of linearizable futures to execute usually takes at least the time needed for executing any subgroup of them (in the best case, it takes the time needed for executing a single linearizable future in the group). Although the inequalities of combining are slightly more restricted than the inequalities of combining and elimination, we believe they are still a reasonable assumption in many cases in practice, as efficient elimination is feasible only for specific data structures in specific situations.

4.3 Lazy Work Stealing

Given a program with futures, we still need a scheduler to assign the threads of the program to different processors in order to run them in parallel. Work stealing [20, 7] is the most popular scheduler for programs in the original future-parallel model, achieving very good load balancing and low scheduling overheads. In this section, we propose a modified work stealing scheduler, called *lazy work stealing*, for programs in linearizable-futures model. We will show in the next section that lazy work stealing can make good use of combining and elimination, by proving good theoretical bounds on its performances.

As in the original work stealing scheduler, each processor in lazy work stealing has its own double-ended queue (deque) to store the threads it created that are ready to execute (in fact, the deque stores the next available nodes of those threads). When a processor executing a thread spawns a normal future thread at a fork, it chooses either the current thread or the newly spawned thread to execute, and push the other into the bottom of its deque. When a processor doesn't have a thread to work on, it pops the first thread from the bottom of its deque to execute if the deque is not empty, and otherwise it randomly chooses another processor and *steals* the first thread from the top of that thread's deque to execute. At the beginning of an execution of a program, the main threads of the program are stored in the deques of some arbitrary processors, so active processors can start popping them out to execute. The execution finishes when all threads (all nodes) in the program have been executed.

To deal with linearizable futures and make good use of combining and elimination, lazy work stealing also does the following:

- For each thread t and each shared object o, all the ready linearizable futures on o created by t are stored together in the shared memory, forming a linked list ℓ in the order they were created. (Note that linearizable futures in the same linked list can be created by different processors, as t can be executed by different processors at different times.)

- When a processor executing t spawns a linearizable future f on o at a fork, it always continues to execute the current thread t and appends f to the linked list ℓ of the ready linearizable futures on o created by t. If f is the last linearizable future on o created by t before the first of the touches of all the ready linearizable futures (including f) on o by t, the processor also creates a pointer node pointing to the head of ℓ and pushes the pointer node to the bottom of its deque (see Figure 4.4).

- When the node a processor pops or steals from a deque is a pointer to the linked list of the ready linearizable futures on o created by t, the processor groups those linearizable futures to execute together, with combining and elimination if possible.

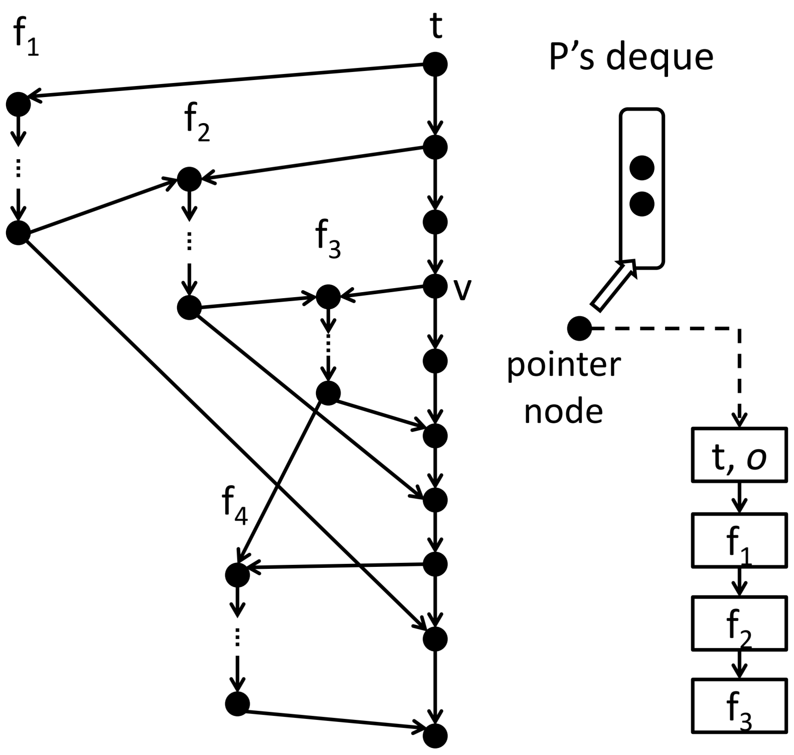


Figure 4.4: An example illustrating how lazy work stealing works when a processor P is executing node v in thread t. First, f_3 is added to the linked list of ready linearizable futures on object o created by thread t. Since f_3 is the last future on o before the first touch of the ready futures on o (i.e., the touch of f_3), a pointer node to the linked list is pushed into the bottom of P's deque.

We can observe that using lazy work stealing, linearizable futures can be executed only if the pointer to the linked list containing them has been pushed into a deque. It is also easy to see that in any execution linearizable futures are always partitioned into the same groups to execute.

Now we explain why a linked link described above can be easily and efficiently implemented. We can, for example, maintain a hash table and use t and o as the key to quickly find the location of the linked list for ready linearizable futures on o created by t. The linked list is created dynamically only when a linearizable future on o is created by t, and its memory can be freed when all the ready linearizable futures have been executed, so lazy work stealing won't take much extra memory in general. Also note that (1) only one processor takes control of thread t at a time and (2) when a processor executes the future group for the ready linearizable futures on o created by t, no more linearizable futures on o will be added to that linked list, as the creation (fork) of the next linearizable future on o created by t is after the touch of that future group. Therefore, we can conclude that only one processor at a time can update the linked list and hence a simple implementation of the linked list without supporting concurrency will suffice.

The only tricky part in the implementation of lazy work stealing is to figure out whether a linearizable future on o is the last one before the first touch of the ready linearizable futures on o in the same thread. To achieve that, we can, for example, have the compiler find out and mark all the linearizable futures of that kind when it compiles the program. Alternatively, we can have the programmer (or the IDE automatically) mark those linearizable futures when the programmer is writing the program. Given that futures and touches are key words explicitly written in the program, Both methods should be easy and convenient.

4.4 Performance Analysis of Lazy Work Stealing

One may concern that lazy work stealing is very inefficient in some scenarios, as lazily waiting for all linearizable futures to be created before executing them can sometimes waste too much CPU time of the idle available processors. In this section, we prove that lazy work stealing performs well in linearizable-futures model, having the best bounds on execution times that any non-clairvoyant scheduler can achieve. Moreover, we prove that in many cases, the performance of lazy work stealing is also close to that of an optimal offline scheduler. Note that in our analysis, we assume each processor can execute different main threads and future threads in an execution, but our results also applies to the case where each processor is assigned to only one main thread and its descendant threads all the time, which is usually the case in hierarchical shared memory systems.

Consider a group g of linearizable futures on object o created by thread t in an execution of a program. We define the *span* of g as the path in thread t from the fork that creates the first future in g to the last touch of the futures in g. The span of g is divided into three segments—the creation span, the middle span, and the touch span. The *creation span* of g is from the the fork that creates the first future in g to the fork that creates the last future in g. The *middle span* of g is from the fork that creates the last future in g to the first touch of the futures in g. The *touch span* of g is from the first touch to the last touch of the futures in g. For instance, in any execution of the program in Figure 4.4 by lazy work stealing, f_1, f_2 and f_3 are grouped, and the creation, middle and touch spans of group $g(f_1 \cup f_2 \cup f_3)$ are the paths from the fork of f_1 to v, from v to the touch of f_3, and from the touch of f_3 to the touch of f_1, respectively, in thread t.

We will prove theoretical bounds for programs in general form and programs in a special form, called *well-formed* programs, in linearizable-futures model. A program in linearizable-futures model is well formed, if in the DAG of any execution by lazy work stealing, the spans of future groups on the same object created by the same thread don't overlap. Intuitively, this means that once a thread touches a linearizable future on object o, it has to touch all the other ready linearizable futures on o it has created, before it can create new linearizable futures on o (see Figure 4.5). It is easy to see that a well-formed program implicitly divides linearizable futures into the same groups as lazy work stealing does.

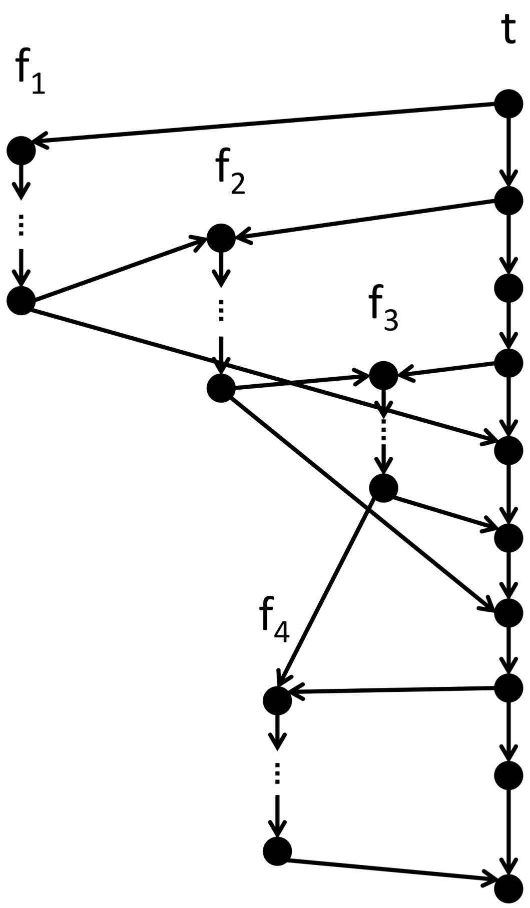


Figure 4.5: A well-formed program obtained by modifying the program in Figure 4.4. The only modification is moving the touch of f_1 to a node before the fork of f_4, so that f_4 is created after all f_1, f_2, and f_3 have be touched.

Recall that in all executions of a program by lazy work stealing, linearizable futures are partitioned into the same groups and all those executions can be represented as the same DAG. We define the *containment level of a program* with respect to the DAG G of its executions by lazy work stealing as follows. Let the size of a future group be the number of linearizable futures in that group. We define the *containment level of the creation span (resp. the touch span) of a group g* in G as the number of different shared objects on which there are future groups of size at least 2 whose middle spans are contained in the creation span (resp. the touch span) of g. The *containment level of g* is the maximum of the containment level of the creation span of g and the containment level of the touch span of g. Finally, the *containment level of a program* is the maximum of the containment levels of all the future groups in G. We will show later in the section that the containment level of a program is often a small constant in practice.

The main results about the performance of lazy work stealing are as follows:

Theorem 28 *Given a program with combining optimization in linearizable-futures model, its execution time by an optimal offline scheduler is $\Theta(\frac{T_1}{P_A} + T_\infty)$, where T_1 and T_∞ are the number of nodes and the length of a critical path in the DAG of the execution respectively, and P_A is the average number of available processors in the execution. Its execution time by lazy work stealing is $O(\frac{T_1}{P'_A} + (c+1)T_\infty)$ in expectation, where c is the containment level of the program and P'_A is the average number of available processors in the execution.*

Theorem 29 *Given a well-formed program with combining and elimination in linearizable-futures model, its execution time by an optimal offline scheduler is $\Theta(\frac{T_1}{P_A} + T_\infty)$, where T_1 and T_∞ are the number of nodes and the length of a critical path in the DAG of the execution respectively, and P_A is the average number of available processors in the execution. Its execution time by lazy work stealing is $O(\frac{T_1}{P'_A} + (c+1)T_\infty)$ in expectation, where c is the containment level of the program and P'_A is the average number of available processors in the execution.*

We start with the proofs of the lemmas for proving Theorem 29.

Lemma 30 *Let T_1 be the number of nodes in the DAG G of any execution of a well-formed program by lazy work stealing. The number of nodes in the DAG of any execution of that program by any scheduler is at least T_1.*

Proof. Consider a linearizable future f on object o created by a main thread t and the group g that f is in in G. Since the program is well formed, any linearizable future on o created by t that is not in g cannot be grouped with f in any execution by any scheduler. This implies any scheduler can either group all the linearizable futures in g to execute together or split them into subgroups to execute one by one. According to the inequalities of combining and elimination, the number of nodes in the g is no more than the number of all the nodes in the those subgroups of g in any execution. Since the number of nodes in normal futures are fixed in all executions and the work stealing scheduler incurs the smallest total number of nodes in linearizable futures, we can conclude that T_1 is the lower bound for any execution by any scheduler. □

The following lemma holds not only for well-formed programs, but also for programs in the general form.

Lemma 31 *Let G be the DAG of an execution of a program of containment level c by lazy work stealing and let $g_1, g_2, ..., g_n$ denote all the future groups contained in a path in G. If $g_1, g_2, ..., g_n$ are on $2c+4$ or more different shared objects, then any linearizable future in g_1 precedes any linearizable future in g_n in the program.*

Proof.

Since $g_1, g_2, ..., g_n$ are on $2c+4$ or more different shared objects, there must be a group g_i whose touch u divides the path into two segments such that the future groups contained in each of the two segments are on at least $c+2$ different shared objects. Note that $g_1, g_2, ..., g_n$ must be spawned and touched by the same main thread. Therefore, we know that the touch of any linearizable future in g_1 must be a node in that main thread before u, because otherwise the touch span of g_1 will contain all the middle spans of $g_2, g_3, ..., g_i$ which are on at least $c+1$ different shared objects, contradicting that the containment level of the program is c. Similarly, the fork of any linearizable future in g_n must be a node in that main thread after u, because otherwise the creation span of g_n will contain all the middle spans of $g_{i+1}, g_{i+2}, ..., g_{n-1}$ which are on at least $c+1$ different shared objects, contradicting that the containment level of the program is c. Therefore, we can conclude that any linearizable future in g_1 must precede any linearizable future in g_n in the program. □

Lemma 32 *Let G be the DAG of an execution of a well-formed program by lazy work stealing, and let $g_1, g_2, ..., g_n$ be all the future groups on object o created by a main thread in G. In the DAG of any execution of the program by any scheduler, there exists a path from u to v such that the path is not shorter than the sum of the lengths of $g_a, g_{a+1}, ..., g_b$ for any $1 \leq a \leq b \leq n$, where u is the fork of a linearizable future in g_a and v is the touch of a linearizable future in g_b.*

Proof. As we pointed out in the proof of Lemma 30, since the program is well-formed, for any $1 \leq i \leq n$, the linearizable futures in g_i have to either be grouped together or be partitioned into subgroups of g_i in any execution by any scheduler. By the inequalities of combining and elimination, the sum of the lengths of those subgroups is not smaller than the length of g_i. Therefore in any execution by any scheduler, the groups consisting of all the linearizable futures in $g_a, g_{a+1}, ..., g_b$ form a path $\alpha_{u,v}$ from the fork u of the first group to the touch v of the last group, through all the nodes in those groups (with order edges connecting successive groups), and the length of $\alpha_{u,v}$ is not shorter than the sum of the lengths of $g_a, g_{a+1}, ..., g_b$. It is obvious that u is the fork of a linearizable future in g_a and v is the touch of a linearizable future in g_b, so $\alpha_{u,v}$ suffices. □

Lemma 33 *Let G be the DAG of an execution of a well-formed program of containment level c by lazy work stealing. If the length of a critical path in G is T_∞, then there is a path of length $\Omega(\frac{T_\infty}{c+1})$ in the DAG of any execution of the program by any scheduler.*

Proof. Consider any critical path in G and all the future groups contained in the path. We partition the critical path into segments as follows. The first segment of the path is from the first node of the path to the touch of the first future group contained in the path, such that the future groups contained in this segment are on $2c+4$ different objects. Then by induction, the next segment is from the end of the previous segment to the touch of the first future group in the rest of the path, such that the future groups contained in this segment are on $2c+4$ different objects. This partition completes when we reach the end of this critical path and hence the last segment is from the end of the previous segment to the end of the path. Let $\beta_1, \beta_2, ..., \beta_n$ denote the segments. Without loss of generality, assume $n = 2k$ is an even number.

Now we consider β_i, for any $1 \leq i \leq n$. By Lemma 32, for an object o on which there are future groups in β_i, there exists a path α_o from u to v in the DAG G' of any execution by any scheduler, such that α_o is not shorter than the sum of the lengths of all the groups on o in β_i in G, where u is the fork of a linearizable future in the first group on o in β_i and v is the touch of a linearizable future in the first group on o in β_i. In G', we can also find a path α' from the start of β_i to the end of β_i, such that α' contains all the segments of β_i that are in normal futures and the main thread, and goes through the nodes in the main thread between the fork and the touch of each linearizable future group instead of going through that group (see Figure 4.6). Since α' contains all the nodes in the normal futures and the main thread in β_i, and for each object o, α_o is not shorter than the sum of the lengths of all the linearizable future groups on o in β_i in G, we know that the sum of the lengths of α' and all the paths α_o is not smaller than the length of β_i. Since the future groups in β_i are on at most $2c+4$ different objects, there is a path α_{β_i} among α' and all α_o, such that α_{β_i} is not shorter than $\frac{1}{2c+5}$ of the length of β_i.

Note that the first node of α' is the first node of β_i, which is essentially the touch of a linearizable future in the last group in β_{i-1}. Also note that the first node of any α_o is the fork of a linearizable future in the first group on o in β_i. Hence, we know the first node of α_{β_i} is either the touch of a linearizable future in the last group in β_{i-1} or the fork of a linearizable future in a group in β_i. Since both the last node of α' (which is the last node of β_i) and the last node of any α_o are the touches of linearizable futures in groups in β_i, we know the last node of α_{β_i} is the touch of a linearizable future in a group in β_i (note that the last node of α_o may be after the last node of β_i).

For any $1 \leq i \leq k-1$, since the future groups in β_{2i} are on $2c+4$ different objects, by Lemma 31, any linearizable future in a group in β_{2i-1} precedes both any linearizable future in the last group in β_{2i} and any linearizable future in a group in β_{2i} in the program. Since (1) the last node of $\alpha_{\beta_{2i-1}}$ is the touch of a linearizable future in a group in β_{2i-1} and (2) the first node of $\alpha_{\beta_{2i+1}}$ is either the touch of a linearizable future in the last group in β_{i-1} or the fork of a linearizable future in a group in β_i, we can conclude that in G', the last node of $\alpha_{\beta_{2i-1}}$ is before the first node of $\alpha_{\beta_{2i+1}}$. Hence, in G', there is always a path α_{odd} containing the paths $\alpha_{\beta_{2i-1}}$ for all $1 \leq i \leq k$ and the length of the path is at least $\frac{1}{2c+5}$ of the sum of the lengths of the paths β_{2i-1} for all $1 \leq i \leq k$. Similarly, we can prove that in G', there is always a path α_{even} containing the paths $\alpha_{\beta_{2i}}$ for all $1 \leq i \leq k$ and the length of the path is at least $\frac{1}{2c+5}$ of the sum of the lengths of the paths β_{2i} for all $1 \leq i \leq k$. Therefore, the sum of the lengths of α_{odd} and α_{even} is at least $\frac{1}{2c+5}$ of the length of the union of β_i for all $1 \leq i \leq 2k$, which is the critical path in G. Hence, at least one of α_{odd} and α_{even} is of length $\Omega(\frac{T_\infty}{c+1})$. □

Proof of Theorem 29. It is a well-known result that any execution of a program by any scheduler

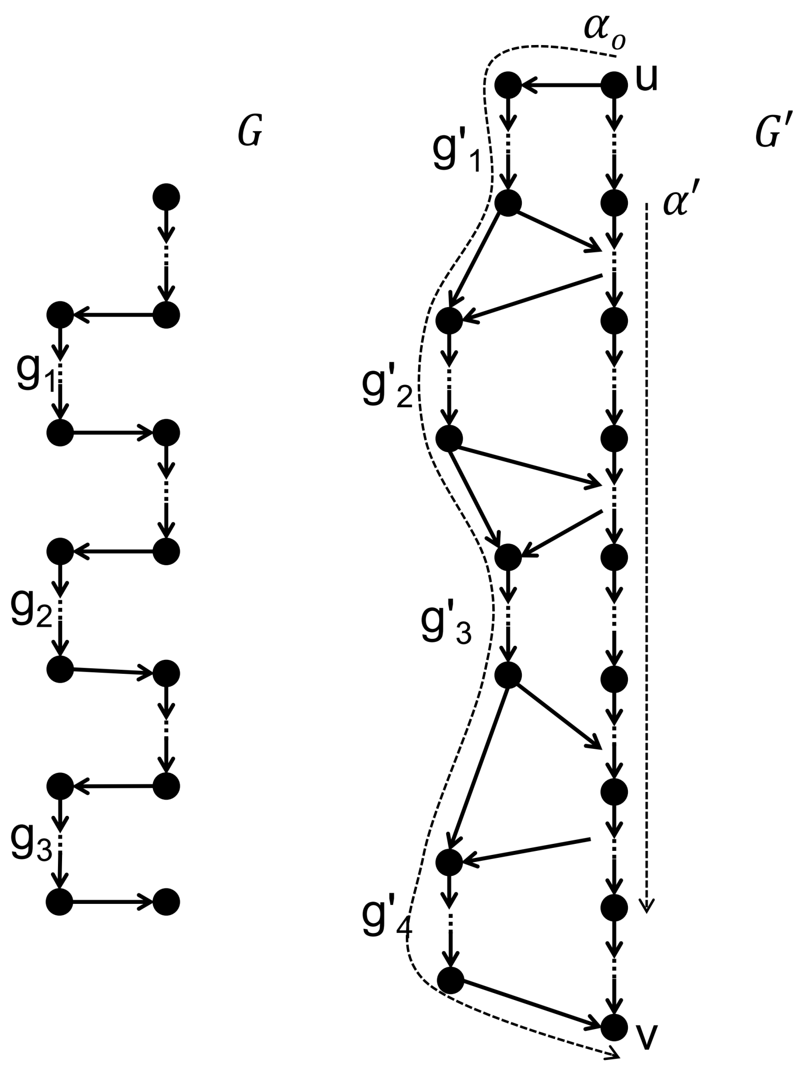


Figure 4.6: The DAG on the left is a path β_i in G. g_1, g_2, and g_3 are linearizable futures on object o contained in β_i, and the other segments of β_i are all in the main thread and normal futures (for simplicity, we assume there is no linearizable futures on other objects). The DAG on the right is part of G', where the dotted paths are α_o and α'.

takes time $\Omega(\frac{T_1^*}{P_A^*} + T_\infty^*)$, where T_1^* and T_∞^* are the number of nodes and the length of a critical path in the DAG of the execution respectively, and P_A^* is the average number of available processors in the execution. It has been proved that the (nonblocking) work stealing scheduler is asymptotically optimal, achieving $\Theta(\frac{T_1^*}{P_A^*} + T_\infty^*)$ execution time in expectation for any program with only normal futures (i.e., a deterministic program whose DAG is fixed) [7]. A key observation is that lazy work stealing behaves the same as the ordinary work stealing scheduler with respect to G, the DAG of any execution of the program by lazy work stealing. This implies that the expected execution time of a program in our model by lazy work stealing is $\Theta(\frac{T_1'}{P_A'} + T_\infty')$, where T_1' and T_∞' are the number of nodes and the length of a critical path in G, respectively.

By Lemmas 30 and 33, in the DAG of any execution of the program by an optimal offline scheduler, the number of nodes T_1 is at least T'_1 and the length of a critical path T_∞ is $\Omega(\frac{T'_\infty}{c+1})$, where c is the containment level of the program. Hence, we can conclude that the expected execution time of the program by lazy work stealing is $O(\frac{T_1}{P'_A} + (c+1)T_\infty)$. □

Now we prove Theorem 28 with the help of the lemmas below.

Lemma 34 *Let G be the DAG of any execution of a program by lazy work stealing, and let $g_1, g_2, ..., g_n$ be all the future groups on an object o created by a main thread in G. For any $1 \leq a \leq b \leq n$, let f_a and f_b denote the first future in g_a and the last future in g_b, respectively. In the DAG G' of any execution of the program by any scheduler, the groups $g'_{a'}, g'_{a'+1}, ..., g'_{b'}$ that contain all the futures in $g_a, g_{a+1}, ..., g_b$ satisfy $T(g'_{a'}) + T(g'_{a'+1}) + ... + T(g'_{b'}) \geq \frac{1}{2}(T(g_a) + T(g_{a+1}) + ... + T(g_b))$, where $g'_{a'}$ contains f_a and $g'_{b'}$ contains f_b.*

Proof. We first prove that no group in G' can contain both a linearizable future in g_i and a linearizable future in g_j, for any $1 \leq i, j \leq n$ and $j \geq i + 2$. To see that, suppose by way of contradiction that a group g' contains both a linearizable future in g_i and a linearizable future in g_j. By medium futures linearizability, g' must contain all the linearizable futures in g_{i+1}. However, we know that the touch of some linearizable future in g_{i+1} is before the fork of any linearizable future in g_j (because of the way work stealing groups linearizable futures), contradicting that g' can contain a linearizable future in g_j.

Without loss of generality, suppose $b = a + 2k$ for some $k \geq 0$. For any $a \leq i \leq b$, let S'_i denote the set of groups in G' that contain linearizable futures in g_i. What we just proved above implies that $S'_i \cap S'_j = \emptyset$ for any $j \geq i + 2$. Let $S'_{a+2i} = \{g'_c, g'_{c+1}, ..., g'_d\}$ be the subset of $\{g'_{a'}, g'_{a'+1}, ..., g'_{b'}\}$, for some $0 \leq i \leq k$, where g'_c and g'_d contain the first and the last linearizable futures in group g_{a+2i}, respectively. By the inequalities of combining and elimination, we have $\sum_{g' \in S'_{a+2i}} T(g') = T(g'_c) + T(g'_{c+1}) + ... + T(g'_d) \geq T(g^*_c) + T(g'_{c+1}) + T(g'_{c+2}) + ... + T(g'_{d-1}) + T(g^*_d) \geq T(g_{a+2i})$, where $T(g^*_c)$ and $T(g^*_d)$ are the subgroups of $T(g'_c)$ and $T(g'_d)$, respectively, that contain only the futures in g_{a+2i}. Since $S'_{a+2i} \cap S'_{a+2j} = \emptyset$ for any $i \neq j$, we have

$$\sum_{i=0}^{k} \sum_{g' \in S'_{a+2i}} T(g') \geq \sum_{i=0}^{k} T(g_{a+2i}).$$

Similarly, we can prove

$$\sum_{i=0}^{k-1} \sum_{g' \in S'_{a+2i+1}} T(g') \geq \sum_{i=0}^{k-1} T(g_{a+2i+1}).$$

Therefore, we can conclude that either the total number of nodes in the future groups in $\bigcup_{i=0}^{k} S'_{a+2i}$ or the total number of nodes in the future groups in $\bigcup_{i=0}^{k-1} S'_{a+2i+1}$ is not smaller than $\frac{1}{2}(T(g_a) + T(g_{a+1}) + ... + T(g_b))$. Since both the future groups in $\bigcup_{i=0}^{k} S'_{a+2i}$ and the future groups in $\bigcup_{i=0}^{k-1} S'_{a+2i+1}$ are among $g'_{a'}, g'_{a'+1}, ..., g'_{b'}$, we complete the proof. □

Lemma 35 *Let T_1 be the number of nodes in the DAG of any execution of a program by lazy work stealing. Then the number of nodes in the DAG of an execution of the program by any scheduler is at least $T_1/2$.*

Proof. Lemma 34 implies that in the DAG of any execution of the program by any scheduler, the total number of nodes in linearizable groups on an object o created by a main thread is at least half of that in the DAG of an execution by lazy work stealing. Given that the total number of nodes in the main threads and the normal futures in the program is fixed in all executions, we can conclude that the number of nodes in the DAG of any execution by any scheduler is at least $T_1/2$. □

Lemma 36 *Let G be the DAG of an execution of a program by lazy work stealing, and let $g_1, g_2, ..., g_n$ be all the future groups on object o created by a main thread in G. In the DAG of any execution of the program by any scheduler, there exists a path from u to v such that the path is not shorter than half of the sum of the lengths of $g_a, g_{a+1}, ..., g_b$ for any $1 \leq a \leq b \leq n$, where u is after the fork of some linearizable future in g_a and v is before the touch of some linearizable future in g_b.*

Proof. By Lemma 34, we know the path from the fork of g'_a to the touch of g'_b through all the nodes in groups $g'_c, g'_{c+1}, ..., g'_d$ is not shorter than half of the sum of the lengths of $g_a, g_{a+1}, ..., g_b$. Now the only thing we need to prove is that the fork of g'_c is after the fork of some linearizable future in g_a and g'_d is before the touch of some linearizable future in g_b. This follows from the fact that g'_c contains the first future in g_a and g'_d contains the last future in g_b. □

Lemma 37 *Let G be the DAG of an execution of a program of containment level c by lazy work stealing. If the length of a critical path in G is T_∞, then there is always a path of length $\Omega(\frac{T_\infty}{c+1})$ in the DAG of any execution of the program by any scheduler.*

Proof. The proof is almost identical to that of Lemma 32. The only difference is this proof is based on Lemma 36, instead of Lemma 32. □

Proof of Theorem 28. The proof is almost identical to that of Theorem 29. □

We argue that many programs in practice have small containment levels. For example, if a program makes operation calls to only c different shared objects, for some small constant c, its containment level is at most c. Moreover, since containment level is only related to the interleaving of linearizable futures in a very restricted form—the creation or touch span of a future group of size at least 2 containing the middle span of another future group of size at least 2, interleaving in other forms doesn't affect the containment level of a program at all. For instance, a program can have a very small containment level even if (1) a thread creates a large number of linearizable futures first and then touches them later, since their interleaving is not in the form related to the containment level, (2) the creation/touch span of a group contains the middle spans of a lot of groups that are only on a few different shared objects, and (3) the creation/touch span of a group contains the middle spans of a lot of groups of size 1 (i.e., single linearizable futures).

The theorem below shows that no non-clairvoyant scheduler can achieve better bounds than lazy work stealing. A scheduler is non-clairvoyant if it cannot know the part of the DAG that has not been executed yet. Here we give a non-clairvoyant scheduler extra power to know whether a linearizable future on object o is the last one in a thread before the first touch of the ready linearizable futures on o in the thread, as lazy work stealing knows that information.

Theorem 38 *Given a non-clairvoyant scheduler $\mathcal{A}$, an integer $c \geq 0$, and the set of available processors at each step of time, there exists a well-formed program of containment level at most c with combining optimization in linearizable-futures model, such that its execution time using an optimal offline scheduler is $\Theta(\frac{T_1}{P_A} + T_\infty)$ and its execution time using $\mathcal{A}$ is $\Omega(\frac{T_1}{P'_A} + (c+1)T_\infty)$ in expectation, where P_A and P'_A are the average numbers of available processors in the two executions respectively, and T_1 and T_∞ are the number of nodes and the length of a critical path in the DAG of the execution by the optimal offline scheduler respectively.*

Theorem 38 (the existence of such a well-formed program with only combining) covers both Theorem 28 (a program with only combining) and Theorem 29 (a well-formed program with combining and elimination), and hence the upper bounds in the two theorems are tight. Its proof is presented below.

Proof of Theorem 38. When $c = 0$, the theorem is trivially true. Now we consider $c > 0$. Since $\mathcal{A}$ is non-clairvoyant, we will adaptively construct a program Q, based on the probabilities of the choices $\mathcal{A}$ has made in the execution of the part of Q that has been generated, such that the expected number of nodes and the expected length of a critical path in the DAG of an execution of G by $\mathcal{A}$ are $\Omega(T_1)$ and $\Omega(cT_\infty)$, respectively.

We construct Q in c steps. In the first step, we construct Q_1^*, a subgraph of Q. we first let $\mathcal{A}$ execute a program Q_1, whose DAG is shown in Figure 4.7. Q_1 has only one main thread, in which c linearizable futures $f_{1,1}, f_{1,2}, ..., f_{1,c}$, all on object o_1, are created consecutively and then touched consecutively in the same order as they are created. We assume that grouping any linearizable futures takes time t, for some fixed number t, that is, $T(g(f_{1,i} \cup f_{1,i+1} \cup ... \cup f_{1,j})) = T(g(f_{1,i})) = t$, for any $1 \leq i \leq j \leq c$.

Let $P_{1,i}$ be the probability that $\mathcal{A}$ groups $f_{1,i}$ with some other linearizable futures in an execution of Q_1, for any i. If $P_{1,i} < 1/2$ for all $1 \leq i \leq c$, we know that in expectation at least $c/2$ linearizable futures are executed solely with no other futures in an execution by $\mathcal{A}$. Therefore, the length of any path ℓ_1 going though all the groups containing the c futures in the DAG of an execution by $\mathcal{A}$ is at least $tc/2$ in expectation.

Now consider the case where $P_{1,k} \geq 1/2$ for some $1 \leq k \leq c$. Since at least one of $f_{1,k-1}$ and $f_{1,k+1}$ must be in the group containing $f_{1,k}$ when $f_{1,k}$ is grouped with other futures, we know that the probability that $f_{1,k'}$ and $f_{1,k'+1}$ are in the same group in an execution by $\mathcal{A}$ is at least $1/4$, for some $k' = k-1$ or $k' = k$. Now let us construct another program Q'_1 by modifying Q_1 as follows. The positions of the touches of $f_{1,k'+1}, f_{1,k'+2}, ..., f_{1,c}$ are all moved to a super node S (which represents a sequence of touch nodes) at the end of the complete program Q, as illustrated in Figure 4.7.

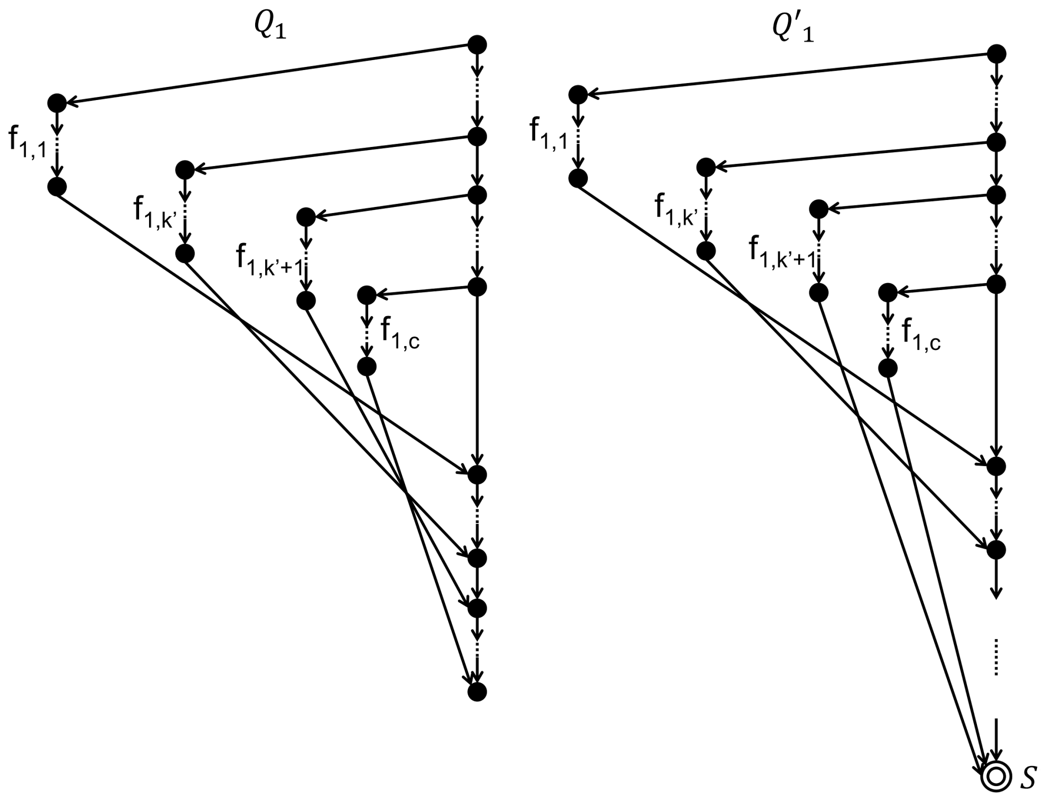


Figure 4.7: Q_1 and Q_1' in the proof of Theorem 38

The execution times of future groups are also modified. We keep $T(g(f_{1,i} \cup f_{1,i+1} \cup ... \cup f_{1,j})) = T(g(f_{1,i})) = t$, for any $1 \leq i \leq j \leq k'$, but we let $T(g(f_{1,i} \cup f_{1,i+1} \cup ... \cup f_{1,j})) = T(g(f_{1,j})) = ct$, for any $j \geq k' + 1$. Intuitively, $f_{1,k'+1}, f_{1,k'+2}, ..., f_{1,c}$ are now "longer" futures each taking time ct, and grouping any of them, with or without linearizable futures prior to $f_{k'+1}$, takes ct. Note that the execution times of the future groups in G_1 and G'_1 all satisfy the inequalities of combining.

A key observation is that the two programs Q_1 and Q'_1 are indistinguishable from $\mathcal{A}$'s perspective before $\mathcal{A}$ starts executing the group containing $f_{1,k'}$, because (1) the touch of $f_{1,k'}$ blocks $\mathcal{A}$ from moving forward in the main thread and (2) all the instructions in the main thread before that touch and the future groups containing only futures prior to $f_{1,k'}$ are the same in the two problems. More specifically, given the same sequence of random bits for running $\mathcal{A}$ (since it may be a randomized scheduler) and the same available processors at each time step, the executions of Q_1 and Q'_1 by $\mathcal{A}$ are identical until $\mathcal{A}$ starts executing the group containing $f_{1,k'}$. Therefore, it is still true that the probability that $f_{1,k'}$ and $f_{1,k'+1}$ are in the same group in an execution of Q'_1 by $\mathcal{A}$ is at least $1/4$. Thus in an execution of Q'_1 by $\mathcal{A}$, the expected length of the path ℓ'_1 consisting of the group containing $f_{1,k'}$ is at least $ct/4$ and obviously the end of ℓ'_1, which is the touch of that group, is the touch of a future $f_{1,i}$ for some $i \leq k'$.

If $P_{1,k} \geq 1/2$ for some $1 \leq k \leq c$, we set $Q^*_1 = Q_1$ and $\ell^*_1 = \ell_1$, and otherwise we set $Q^*_1 = Q'_1$ and $\ell^*_1 = \ell'_1$. Thus, in the DAG of an execution of Q^*_1 by $\mathcal{A}$, the expected length of ℓ^*_1 is at least $ct/4$.

Now in the second step of constructing Q, we generate Q^*_2 by extending Q^*_1 as follows. Starting from the "last node" of Q^*_1 (which is either the touch of $f_{1,c}$ in Q_1 or the touch of $f_{1,k'}$ in Q'_1), we create linearizable futures $f_{2,k'+1}, f_{2,k'+2}, ..., f_{2,c}$ on another object o_2 consecutively in the main thread and then touch them in the same order. We assume grouping any of those futures takes time t. Let Q_2 denote the new program that consists of $Q{*}_1$ and the newly added part. Like what we did for Q_1, if the probability that $f_{2,i}$ is executed solely without other linearizable futures in an execution by $\mathcal{A}$ is less than $1/2$ for all i, we let $Q^*_2 = Q_2$. Otherwise, there must be futures $f_{2,k'}$ and $f_{2,k'+1}$ for some k' such that they are in the same group in an execution by $\mathcal{A}$ with probability at least $1/4$ and hence we let $Q^*_2 = Q'_2$, where Q'_2 is obtained by moving the touches of $f_{2,k+1}, f_{2,k+2}, ..., f_{2,c}$ to the super node S and changing the execution time to ct for any group containing any of $f_{2,k+1}, f_{2,k+2}, ..., f_{2,c}$. Again, we can show that Q_2 and Q'_2 are indistinguishable from $\mathcal{A}$'s perspective until $\mathcal{A}$ starts executing the group containing $f_{k'}$. By the same argument for Q^*_1, We can prove that there is a path ℓ^*_2 from the fork of $f_{2,i}$ to the touch of $f_{2,j}$, for some i, j, such that its length is at least $ct/4$ in expectation in the DAG of an execution of Q^*_2 by $\mathcal{A}$.

By induction, in the kth step for any $2 \leq k \leq c$, we construct $Q{*}_i$ by "appending" to $Q{*}_{k-1}$ linearizable futures $f_{k,1}, f_{k,2}, ..., f_{k,c}$ on a new object o_k in the same way we explained above, and we can conclude that there is a path ℓ^*_k from the fork of $f_{k,i}$ to the touch of $f_{k,j}$, for some i, j, such that its length is at least $ct/4$ in expectation in the DAG of an execution of Q^*_k by $\mathcal{A}$.

Finally, we construct G by appending the super node S to the last node in the main thread in G^*_c. Noth that S represents a sequence of at most c nodes that are touches of linearizable futures in G_c^*. Now consider any path ℓ that contains all $\ell_1^*, \ell_2^*, ..., \ell_c^*$ in the DAG of an execution of G by $\mathcal{A}$. Obviously, the expected length of ℓ is at least $c \cdot ct/4 = c^2t/4$. In contrast, an optimal offline scheduler will group all $f_{k,1}, f_{k,2}, ..., f_{k,c}$ together if $G_k^* = G_k$, or group all the "shorter" futures $f_{k,1}, f_{k,2}, ..., f_{k,i}$ and then group all the "longer" futures $f_{k,i+1}, f_{k,i+2}, ..., f_{k,c}$ if $G_k^* = G'_k$ for any $1 \leq k \leq c$. It is easy to see that the DAG of an execution by this optimal scheduler has $\Theta(c^2t)$ nodes, asymptotically optimal, and the length of its critical path T_∞ is only $\Theta(ct)$. Thus, the expected length of a critical path in an execution by $\mathcal{A}$ is $\Omega(cT_\infty)$, i.e. $\Omega((c+1)T_\infty)$, which completes the proof. (Note that G has only one main thread. We can also construct a program with multiple main threads that suffices, where each main thread is constructed in the same way as G is constructed.)

□

Chapter 5

Concurrent Data Structures for Near-Memory Computing

In this chapter, we will discuss applications of linearizable futures in the near-memory computing model, also called the PIM model. As we briefly mentioned earlier, to be competitive with traditional concurrent data structures, data structures in the PIM model need new approaches to leverage parallelism. Here we present some PIM-managed concurrent data structures, where threads send their operation requests as linearizable futures to PIM cores which execute those requests with certain optimizations. With those optimizations and the help of PIM cores' fast memory access speed, our data structures can beat state-of-the-art concurrent data structures in the literature.

This chapter is organized as follows. We first present related work in Section 5.1. In Section 5.2 we briefly describe our assumptions about the hardware architecture. In Section 5.3 we introduce a simplified performance model that we use throughout this paper to predict performance of our algorithms using the hardware architecture described in Section 5.2. Finally in Sections 5.4 and 5.5, we describe and analyze our PIM algorithms and use our model to compare them to prior work. We also use current architectures to simulate the behavior of our algorithms and evaluate compared to state-of-the-art concurrent algorithms.

5.1 Related Work

The PIM model is undergoing a renaissance. Studied for decades (e.g., [74, 54, 36, 68, 67, 52, 40]), this model has recently re-emerged due to advances in 3D-stacked techology that can stack memory dies on top of a logic layer [51, 59, 12]. For example, Micron and others have recently released a PIM prototype called the Hybrid Memory Cube [26], and the model has again become the focus of architectural research. Different PIM-based architectures have been proposed, either for general purposes or for specific applications [5, 4, 79, 50, 11, 6, 10, 9, 21, 80, 81].

The PIM model has many advantages, including low energy consumption and high bandwidth (e.g., [4, 79, 80, 9]). Here, we focus on one more: low memory access latency [59, 50, 11]. To our knowledge, however, we are the first to utilize PIM memory for designing efficient concurrent data structures. Although some researchers have studied how PIM memory can help speed up concurrent operations to data structures, such as parallel graph processing [4] and parallel pointer chasing on linked data structures [50], the applications they consider require very simple, if any, synchronization between operations. In contract, operations to concurrent data structures can interleave in arbitrary orders, and therefore have to correctly synchronize with one another in all possible situations. This makes designing concurrent data structures with correctness guarantees like linearizability [49] very challenging.

Moreover, no one has ever compared the performance of data structures in the PIM model with that of state-of-the-art concurrent data structures in the classic shared memory model. We analyze and evaluate concurrent linked-lists and skip-lists, as representatives of pointer-chasing data structures, and concurrent FIFO queues, as representatives of contended data structures. For linked-lists, we compare our PIM-managed implementation with well-known approaches such as fine-grained locking [43] and flat combining [44, 30].

For skip-lists, we compare our implementation with the lock-free skip-list [48] and a skip-list with flat combining and partitioning optimization. For FIFO queues, we compare our implementation with the flat-combining FIFO queue [44] and the F&A-based FIFO queue [64].

5.2 Hardware Architecture and Model

In the PIM hardware model, multiple CPUs are connected to the main memory, via a shared crossbar network, as illustrated in Figure 5.1. The main memory consists of two parts—one is a normal DRAM accessible by CPUs and the other, called the *PIM memory*, is divided into multiple partitions, called *PIM vaults* or simply vaults. According to the *Hybrid Memory Cube* specification 1.0 [26], each HMC consists of 16 or 32 vaults and has total size 2GB or 4 GB (so each vault has size roughly 100MB). We assume the same specifications in our PIM model, although the size of a PIM memory and the number of its vaults can be greater. Each CPU also has access to a hierarchy of caches backed by DRAM, and there can be last-level caches shared among multiple CPUs.

Each vault has a *PIM core* directly attached to it. we say a vault is *local* to the PIM core attached to it, and vice versa. A PIM core is a lightweight CPU that may be slower than a full-fledged CPU with respect to computation speed.[1] A vault can be accessed only by its local PIM core.[2] Although a PIM core is relatively slow computationally, it has fast access to its local vault.

A PIM core communicates with other PIM cores and CPUs via messages. Each PIM core, as

[1]A PIM core can be thought of as an in-order CPU with only small private L1 cache and without some optimizations that full-fledged CPUs usually have.

[2]We may alternatively assume that a PIM core has direct access to remote vaults, at a larger cost. We may also assume that vaults are accessible by CPUs as well, but at the cost of dealing with cache coherence between CPUs and PIM cores. Some cache coherence mechanisms for PIM memory claim to be not costly (e.g., [21, 5]). However, we prefer to keep the hardware model simple and we will show that we are still able to design efficient concurrent data structure algorithms with this simple, less powerful PIM memory.

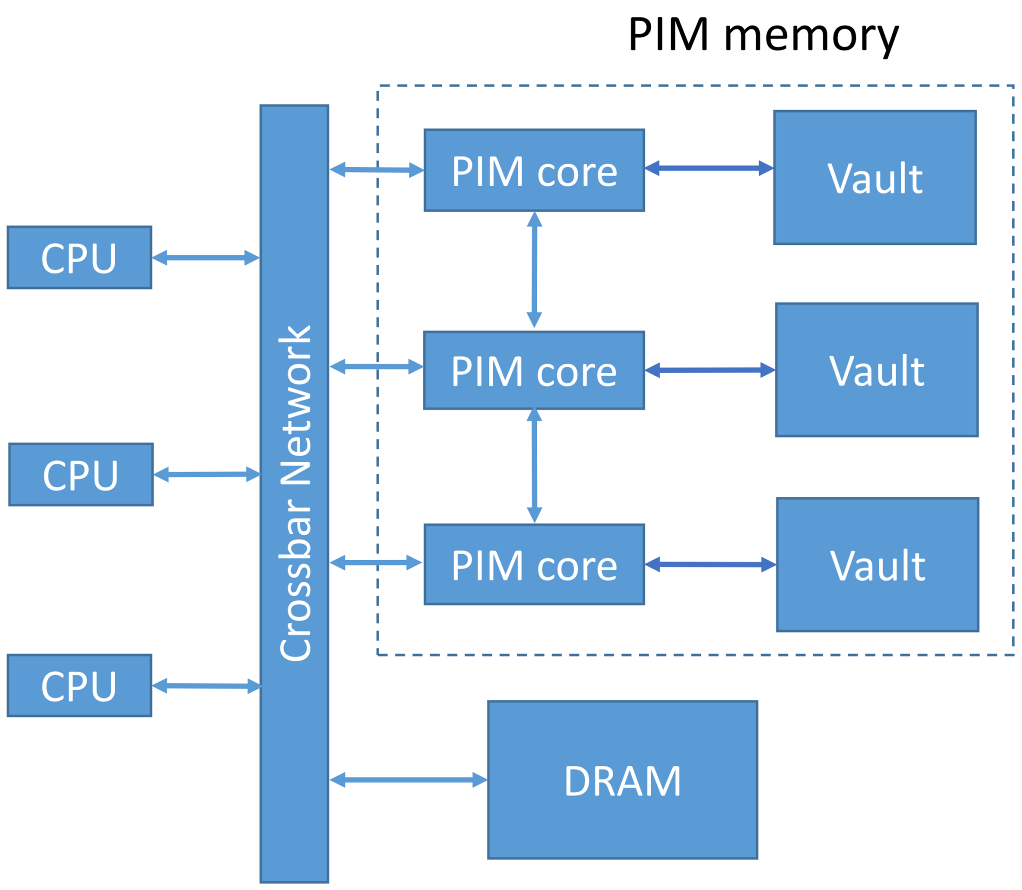


Figure 5.1: The PIM model

well as each CPU, has buffers for storing incoming messages. A message is guaranteed to eventually arrive at the buffer of its receiver. Messages from the same sender to the same receiver are delivered in FIFO order: the message sent first arrives at the receiver first. However, messages from different senders or to different receivers can arrive in an arbitrary order.

To keep the PIM memory simple, we assume that a PIM core can only make read and write operations to its local vault, while a CPU also supports more powerful atomic operations, such as CAS and F&A. Virtual memory is cheap to be achieved in this model, by having each PIM core maintain its own page table for its local vault [50].

5.3 Performance Model

Based on the latency numbers in prior work on PIM memory, in particular on the Hybrid Memory Cube [26, 11], and on the evaluation of operations in multiprocessor architectures [28], we propose the following simple performance model to compare our PIM-managed algorithms with existing concurrent data structure algorithms. For read and write operations, we assume

$$\mathcal{L}_{cpu} = 3\mathcal{L}_{pim} = 3\mathcal{L}_{llc},$$

where $\mathcal{L}_{cpu}$ is the latency of a memory access by a CPU, $\mathcal{L}_{pim}$ is the latency of a local memory access by a PIM core, and $\mathcal{L}_{llc}$ is the latency of a last-level cache access by a CPU. We ignore the costs of cache accesses of other levels in our performance model, as they are negligible in the concurrent data structure algorithms we will consider. We assume that the latency of a CPU making an atomic operation, such as a CAS or a F&A, to a cache line is

$$\mathcal{L}_{atomic} = \mathcal{L}_{cpu},$$

even if the cache line is currently in cache. This is because an atomic operation hitting the cache is usually as costly as a memory access by a CPU, acorrding to [28]. When there are k atomic operations competing for a cache line concurrently, we assume that they are executed sequentially, that is, they complete in times $\mathcal{L}_{atomic}, 2\mathcal{L}_{atomic}, ..., k \cdot \mathcal{L}_{atomic}$, respectively.

We assume that the size of a message sent by a PIM core or a CPU is at most the size of a cache line. Given that a message transferred between a CPU and a PIM core goes through the crossbar network, we assume that the latency for a message to arrive at its receiver is

$$\mathcal{L}_{message} = \mathcal{L}_{cpu}.$$

We make a conservative assumption that the latency of a message transferred between two PIM cores is also $\mathcal{L}_{message}$. Note that the message latency we consider here is the transfer time of a message through a message passing channel, that is, the period between the moment when a PIM or a CPU sends off the message and the moment when the message arrives at the buffer of its receiver. We ignore the time spent in other parts of a message passing procedure, such as preprocessing and constructing the message, as it is negligible compared to the time spent in message transfer.

5.4 Low Contention Data Structures

In this section we consider data structures with low contention; pointer chasing data structures, such as linked-lists and skip-lists, fall in this category. These are data structures whose operations need to de-reference a non-constant sequence of pointers before completing. We assume they support operations such as add(x), delete(x) and contains(x), which follow "next node" pointers until reaching the position of node x. When these data structures are too large to fit in CPU caches and access uniformly random keys, they incur expensive memory accesses, which cannot be easily predicted, making the pointer chasing the dominating overhead of these data structures. Naturally, these data structures have been early examples of the benefit of near-memory computing [50], as the entire pointer chase could be performed by the PIM core, and only the final result returned to the application.

However, under the same conditions, these data structures have inherently low contention. Lock-free algorithms [33, 69, 75, 48] have shown that these data structures can scale to hundreds of cores under low contention. Unfortunately, each vault in PIM memory has a single core; as a consequence, prior work has only compared PIM data structures with sequential data structures, not with carefully crafted concurrent data structures.

We analyze linked-lists and skip-lists, and show that the naive PIM data structure in each case cannot outperform the equivalent CPU managed concurrent data structure even for a small number of cores. Next, we show how to use state-of-the art techniques from concurrent computing literature to optimize algorithms for near-memory computing to outperform well-known concurrent data structures.

5.4.1 Linked-lists

We now describe a naive PIM linked-list. The linked-list is stored in a vault, maintained by the local PIM core. Whenever a CPU[3] wants to perform an operation on the linked-list, it sends a request to the PIM core. The PIM core will retrieve the message, execute the operation, and send the result back to the CPU. The PIM linked-list is sequential, as it can only be accessed by one PIM core.

Doing pointer chasing on sequential data structures by PIM cores is not a new idea (e.g., [50, 4]). It is obvious that for a sequential data structure like a sequential linked-list, replacing the CPU with a PIM core to access the data structure will largely improve its performance due to the PIM core's much faster memory access. However, we are not aware of any prior comparison between the performance of PIM-managed data structures and concurrent data structures in which CPUs can make operations in parallel. In fact, our analytical and experimental results will show that the performance of the naive PIM-managed linked-list is much worse than that of the concurrent linked-list with fine-grained locks [43].

To improve the performance of the PIM-managed linked-list, we apply the following *combining optimization* to it: the PIM core retrieves all pending requests from its buffer and executes all of them during only one traversal over the linked-list. It is not hard to see that the role of the PIM core in our PIM-managed linked-list is very similar to that of the combiner in a concurrent linked-list implemented using *flat combining* [44], where, roughly speaking, threads compete for a "combiner lock" to become the combiner, and the combiner will take over all operation requests from other threads and execute them. Therefore, we think the performance of the flat-combining linked-list is a good indicator of the performance of our PIM-managed linked-list.

Based on our performance model, we can calculate the approximate expected throughputs of the linked-list algorithms mentioned above, when there are p CPUs making operation requests concurrently. We assume that a linked-list consists of nodes with integer keys in the range of $[1, N]$. Initially a linked-list has n nodes with keys generated independently and uniformly at random from $[1, N]$. The keys of the operation requests are generated the same way. To simplify the analysis, we assume that CPUs only make *contains*() requests (or the number of *add*() requests is the same as the number of *delete*() so that the size of each linked-list nearly doesn't change). We also assume that a CPU makes a new operation request immediately after its previous one completes. Assuming

[3]We use the term CPU to refer to CPU cores, as opposed to PIM cores.

that $n \gg p$ and $N \gg p$, the approximate expected throughputs (per second) of the concurrent linked-lists are presented in Table 5.1, where $\mathcal{S}_p = \sum_{i=1}^{n} (\frac{i}{n+1})^p$.[4]

Algorithm	Throughput
Linked-list with fine-grained locks	$\frac{2p}{(n+1)\mathcal{L}_{cpu}}$
Flat-combining linked-list without combining	$\frac{2}{(n+1)\mathcal{L}_{cpu}}$
PIM-managed linked-list without combining	$\frac{2}{(n+1)\mathcal{L}_{pim}}$
Flat-combining linked-list with combining	$\frac{p}{(n-\mathcal{S}_p)\mathcal{L}_{cpu}}$
PIM-managed linked-list with combining	$\frac{p}{(n-\mathcal{S}_p)\mathcal{L}_{pim}}$

Table 5.1: Throughputs of linked-list algorithms.

It is easy to see that the PIM-managed linked-list with combining outperforms the linked-list with fine-grained locks, which is the best one among other algorithms, as long as $\frac{\mathcal{L}_{cpu}}{\mathcal{L}_{pim}} > \frac{2(n-\mathcal{S}_p)}{n+1}$. Given that $0 < \mathcal{S}_p \leq \frac{n}{2}$ and $\mathcal{L}_{cpu} = 3\mathcal{L}_{pim}$, the throughput of the PIM-managed linked-list with combining should be at least 1.5 times the throughput of the linked-list with fine-grained locks. Without combining, however, the PIM-managed linked-list cannot beat the linked-list with fine-grained locks when $p > 6$.

We implemented the linked-list with fine-grained locks and the flat-combining link-list with and without the combining optimization. We tested them on a Dell server with 512 GB RAM and 56 cores on four Intel Xeon E7-4850v3 processors at 2.2 GHz. To get rid of NUMA access effects, we ran experiments with only one processor, which is a NUMA node with 14 cores, a 35 MB shared L3 cache, and a private L2/L1 cache of size 256 KB/64 KB per core. Each core has 2 hyperthreads, for a total of 28 hyperthreads. Cache lines have 64 bytes.

The throughputs of the algorithms are presented in Figure 5.2. The results confirmed the validity of our analysis in Table 5.1. The throughput of the flat-combining algorithm without combining optimization is much worse than the algorithm with fine-grained locks. Since we believe the performance of the flat-combining linked-list is a good indicator of that of the PIM-managed linked-list, we triple the throughput of the flat-combining algorithm without combining optimization to get the estimated throughput of the PIM-managed algorithm. As we can see, it is still far below the throughput of the one with fined-grained locks. However, with the combining optimization, the performance of the flat-combining algorithm improves significantly and the estimated throughput of our PIM-managed linked-list with combining optimization now beats all others'.

5.4.2 Skip-lists

Like the naive PIM-managed linked-list, the naive PIM-managed skip-list keeps the skip-list in a single vault and CPUs send operation requests to the local PIM core that executes those operations.

[4]We define the rank of an operation request to a linked-list as the number of pointers it has to traverse until it finds the right position for it in the linked-list. $\mathcal{S}_p$ is the expected rank of the operation request with the biggest key among p random requests a PIM core or a combiner has to combine, which is essentially the expected number of pointers a PIM core or a combiner has to go through during one pointer chasing procedure.

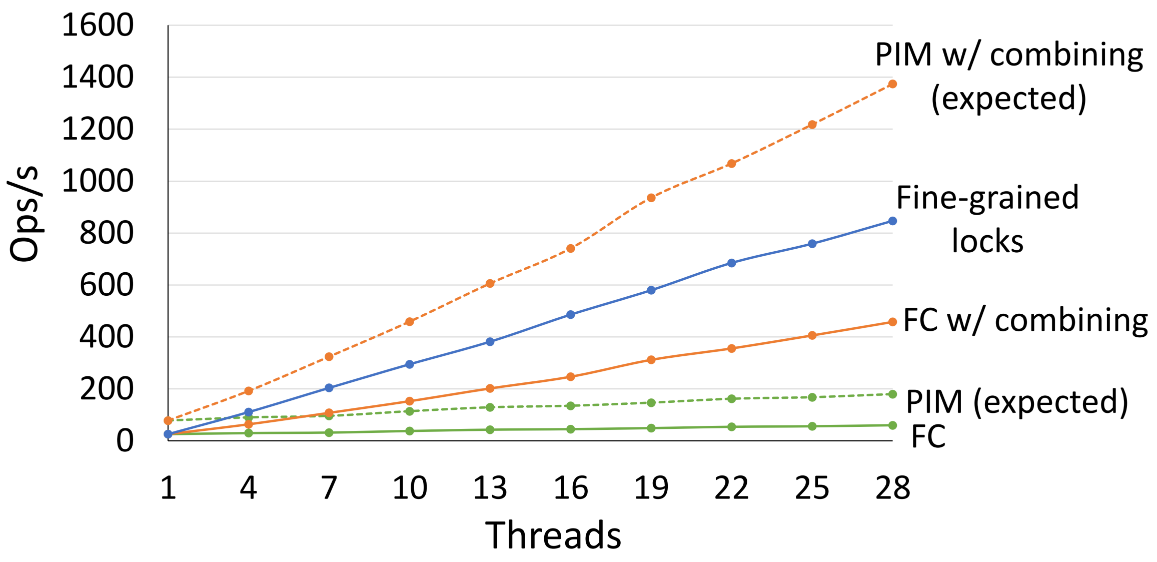


Figure 5.2: Experimental results of linked-lists. We evaluated the linked-list with Fine-grained locks and the flat-combining linked-list (FC) with and without the combining optimization.

As we will see, this algorithm is less efficient than some existing algorithms.

Unfortunately, the combining optimization cannot be applied to skip-lists effectively. The reason is that for any two nodes not close enough to each other in the skip-list, the paths we traverse through to reach them don't largely overlap.

On the other hand, PIM memory usually consists of many vaults and PIM cores. For instance, the first generation of Hybrid Memory Cube [26] has up to 32 vaults. Hence, a PIM-managed skip-list may achieve much better performance if we can exploit the parallelism of multiple vaults. Here we present our PIM-managed skip-list with a *partitioning optimization*: A skip-list is divided into partitions of disjoint ranges of keys, stored in different vaults, so that a CPU sends its operation request to the PIM core of the vault to which the key of the operation belongs.

Figure 5.3 illustrates the structure of a PIM-managed skip-list. Each partition of a skip-list starts with a *sentinel node* which is a node of the max height. For simplicity, assume the max height H_{max} is predefined. A partition covers a key range between the key of its sentinel node and the key of the sentinel node of the next partition. CPUs also store a copy of each sentinel node in the normal DRAM and the copy has an extra variable indicating the vault containing the sentinel node. Since the number of nodes of the max height is very small with high probability, those copies of those sentinel nodes can almost certainly stay in cache if CPUs access them frequently.

When a CPU applies an operation for a key to the skip-list, it first compares the key with those of the sentinels, discovers which vault the key belongs to, and then sends its operation request to that vault's PIM core. Once the PIM core retrieves the request, it executes the operation in the local vault and finally sends the result back to the CPU.

Now let us discuss how we implement the PIM-managed skip-list when the key of each operation

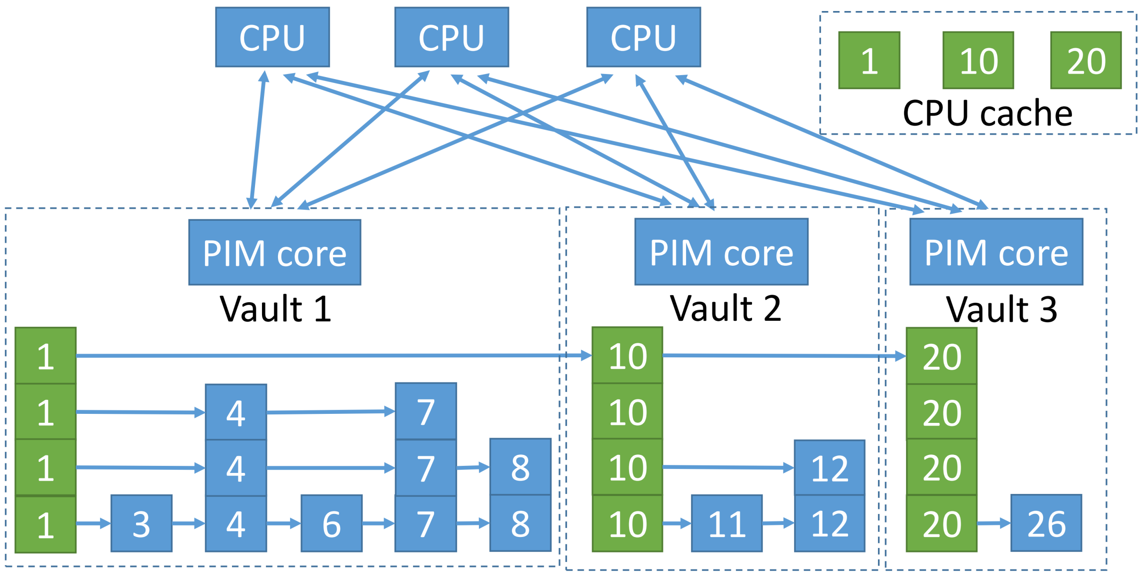


Figure 5.3: A PIM-managed FIFO queue with three partitions

is an integer generated uniformly at random from range $[0, n]$ and the PIM memory has k vaults available. Initially we can create k partitions starting with fake sentinel nodes with keys 0, $1/k$, $2/k$,..., $(n-1)/k$, respectively, and allocate each partition in a different vault. The sentinel nodes will never be deleted. If a new node to be added has the same key as a sentinel node, we insert it immediately after the sentinel node.

We compare the performance of our PIM-managed skip-list with partitions to the performance of a flat-combining skip-list [44] and a lock-free skip-list [48], where p CPUs keeps making operation requests. We also apply the partitioning optimization to the flat-combining skip-list, so that k combiners are in charge of k partitions of the skip-list. To simplify the comparison, we assume that all skip-lists have the same initial structure (expect that skip-lists with partitions have extra sentinel nodes) and all the operations are contains() operations (or the number of *add*() requests is the same as the number of *delete*() so that the size of each skip-list nearly doesn't change). Their approximate expected throughputs are presented in Table 5.2, where β is the average number of nodes an operation has to go through in order to find the location of its key in a skip-list ($\beta = \Theta(\log N)$, where N is the size of the skip-list). Note that we have ignored some overheads in the flat-combining algorithms, such as maintaining combiner locks and publication lists (we will discuss publication lists in more detail in Section 5.5). We also have overestimated the performance of the lock-free skip-list by not counting the CAS operations used in add() and delete() requests, as well as the cost of retries caused by conflicts of updates. Even so, our PIM-managed linked-list with partitioning optimization is still expected to outperform the second best algorithm, the lock-free skip-list when $k > \frac{(\beta\mathcal{L}_{pim}+\mathcal{L}_{message})p}{\beta\mathcal{L}_{cpu}}$. Given that $\mathcal{L}_{message} = \mathcal{L}_{cpu} = 3\mathcal{L}_{pim}$, $k > p/3$ should suffice.

Our experiments have revealed similar results, as presented in Figure 5.4. We have implemented and run the flat-combining skip-list with different numbers of partitions and compared them with

Algorithm	Throughput
Look-free skip-list	$\frac{p}{\beta\mathcal{L}_{cpu}}$
Flat-combining skip-list	$\frac{1}{\beta\mathcal{L}_{cpu}}$
PIM-managed skip-list	$\frac{1}{(\beta\mathcal{L}_{pim}+\mathcal{L}_{message})}$
Flat-combining skip-list with k partitions	$\frac{k}{\beta\mathcal{L}_{cpu}}$
PIM-managed skip-list with k partitions	$\frac{k}{(\beta\mathcal{L}_{pim}+\mathcal{L}_{message})}$

Table 5.2: Throughputs of skip-list algorithms.

the lock-free skip-list. As the number of partitions increases, the performance of the flat-combining skip-list gets better, implying the effectiveness of the partitioning optimization. Again we believe the performance of the flat-combining skip-list is a good indicator to the performance of our PIM-managed skip-list. Therefore, according to the analytical results in Table 5.2, we can triple the throughput of a flat-combining skip-list to get the expected performance of a PIM-managed skip-list. As Figure 5.4 illustrates, when the PIM-managed skip-list has 8 or 16 partitions, it is expected to outperform the lock-free skip-list with up to 28 hardware threads.

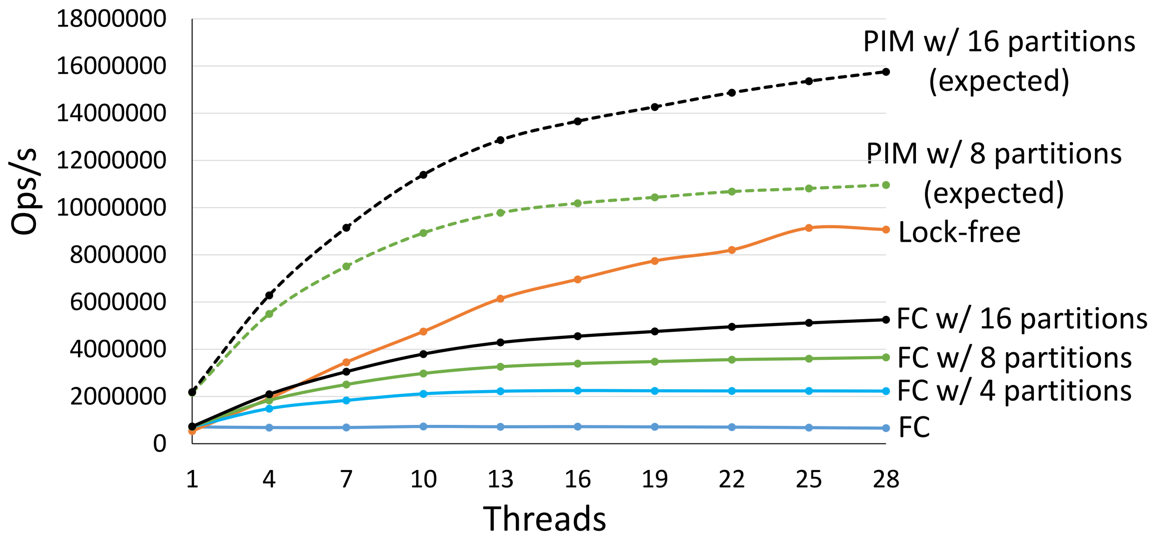


Figure 5.4: Experimental results of skip-lists. We evaluated the lock-free skip-list and the flat-combining skip-list (FC) with different numbers (1, 4, 8, 16) of partitions.

Skip-list Rebalancing

The PIM-managed skip-list performs well with a uniform distribution of requests. However, if the distribution of requests is not uniform, a static partitioning scheme will result in unbalanced partitions, with some PIM cores being idle, while others having to serve a majority of requests. To address this problem, we introduce a non-blocking protocol for migrating consecutive nodes from one vault to another.

The protocol works as follows. A PIM core p that manages a vault v' can send a message to another PIM core q, managing vault v, to request that some nodes are moved from v' to v. First, p sends a message notifying q of the start of the migration. Then p sends messages of adding those nodes to q one by one in an ascending order according to the keys of the nodes. After all the nodes have been migrated, p sends notification messages to CPUs so that they can update their copies of sentinel nodes accordingly. After p receives acknowledgement messages from all CPUs, it notifies q of the end of migration. To keep the node migration protocol simple, we don't allow q to move those nodes to another vault again until p finishes its node migration.

During the node migration, p can still serve requests from CPUs. Assume that a request with key k_1 is sent to p when p is migrating nodes in a key range containing k_1. If p is about to migrate a node with key k_2 at the moment and $k_1 \geq k_2$, p serves the request itself. Otherwise, p must have migrated all nodes in the subset containing key k_1, and therefore p forwards the request to q which will serve the request and respond directly to the requesting CPU.

The algorithm is correct, because a request will eventually reach the vault that currently contains nodes in the key range that the request belongs to: If a request arrives to p which no longer holds the partition the request belongs to, p can simply reply with a rejection to the CPU and the CPU will resend its request to the correct PIM core, because it has already updated its sentinels and knows which PIM core it should contact now.

Using this node migration protocol, the PIM-managed FIFO queue can support two rebalancing schemes: 1) If a partition has too many nodes, the local PIM core can send nodes in a key range to a vault that has fewer nodes; 2) If two consecutive partitions are both small, we can merge then by moving one to the vault containing the other.

In practice, we expect that rebalancing will not happen very frequently, so its overhead can be ameliorated by the improved efficiency resulting from a rebalance.

5.5 High Contention Data Structures

In this section, we consider data structures that are often contended when accessed by many threads concurrently. In these data structures, operations compete for accessing one or several locations, creating a contention spot, which can become a performance bottleneck. Examples include head and tail pointers in queues or the top pointer of a stack.

These data structures have good locality and the contention spots are often found in shared CPU caches, such as the last level cache in a multi-socket non-uniform memory access machine when accessed by threads running only on one socket. Therefore, these data structures might seem to be a poor fit for near-memory computing, because the advantage of the faster access to memory is muted by having the frequently accessed data in the cache. However, such a perspective does not consider the overhead introduced by contention in a concurrent data structure where all threads try to access the same locations.

As a representative example of this class of data structures, we consider a FIFO queue, where concurrent enqueue and dequeue operations compete for the head and tail of the queue, respectively. Although a naive PIM FIFO queue is not a good replacement for a well crafted concurrent FIFO queue, we show that, counterintuitively, PIM can still have benefits over a traditional concurrent FIFO queue. In particular, we exploit the pipelining of requests from CPUs, which can be done very efficiently in PIM, to design a PIM FIFO queue that can outperform state-of-the-art concurrent FIFO queues, such as the one using flat combining [44] and the one using Fetch And Add [64].

5.5.1 FIFO queues

The structure of our PIM-managed FIFO queue is shown in Figure 5.5. A queue consists of a sequence of segments, each containing consecutive nodes of the queue. A segment is allocated in a PIM vault, with a head node and a tail node pointing to the first and the last nodes of the segment, respectively. A vault can contain multiple (mostly likely non-consecutive) segments. There are two special segments—the *enqueue segment* and the *dequeue segment*. To enqueue a node, a CPU sends an enqueue request to the PIM core of the vault containing the enqueue segment. The PIM core will then insert the node to the head of the segment. Similarly, to dequeue a node, a CPU sends a dequeue request to the PIM core of the vault holding the dequeue segment. The PIM core will then pop out the node at the tail of the dequeue segment and send the node back to the CPU.

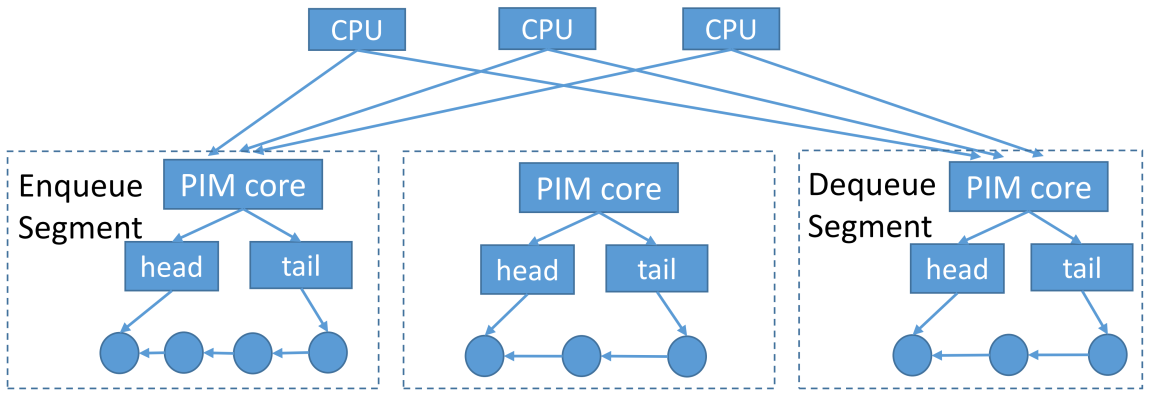


Figure 5.5: A PIM-managed FIFO queue with three segments

Initially the queue consists of an empty segment which acts as both the enqueue segment and the dequeue segment. When the length of enqueue segment exceeds some threshold, the PIM core maintaining it notifies another PIM core to create a new segment as the new enqueue segment.[5] When the dequeue segment becomes empty and the queue has other segments, the dequeue segment is deleted and the segment that was created first among all the remaining segments is designated

[5]When and how to create a new segment can be decided in other ways. For example, CPUs, instead of the PIM core holding the enqueue segment, can decide when to create the new segment and which vault to hold the new segment, based on more complex criteria (e.g., if a PIM core is currently holding the dequeue segment, it will not be chosen for the new segment so as to avoid the situation where it deals with both enqueue and dequeue requests). To simplify the description of our algorithm, we omit those variants.

as the new dequeue segment. (It is not hard to see that the new dequeue segment were created when the old dequeue segment acted as the enqueue segment and exceeded the length threshold.) If the enqueue segment is different from the dequeue segment, enqueue and dequeue operations can be executed by two different PIM cores in parallel, which doubles the throughput compared to a straightforward queue implementation held in a single vault.

Algorithm 2 PIM-managed FIFO queue: PIM core's procedures upon receiving requests enq(cid, u), deq(cid), newEnqSeg(), and newEnqDeq()

```
 1: procedure enq(cid, $u$)
 2:     if enqSeg == null then
 3:         send message(cid, false);
 4:     else
 5:         if enqSeg.head $\neq$ null then
 6:             enqSeg.head.next = $u$;
 7:             enqSeg.head = $u$;
 8:         else
 9:             enqSeg.head = $u$;
10:             enqSeg.tail = $u$;
11:         enqSeg.count = enqSeg.count + 1;
12:         send message(cid, true);
13:         if enqSeg.count > threshold then
14:             cid′ = the CID of the PIM core chosen to maintain the new segment;
15:             send message(cid′, newEnqSeg());
16:             enqSeg.nextSegCid = cid′;
17:             enqSeg = null;
```

```
 1: procedure deq(cid)
 2:     if deqSeg == null then
 3:         send message(cid, false);
 4:     else
 5:         if deqSeg.tail $\neq$ null then
 6:             send message(cid, deqSeg.tail);
 7:             deqSeg.tail = deqSeg.tail.next;
 8:         else
 9:             if deqSeg == enqSeg then
10:                 send message(cid, null);
11:             else
12:                 send message(deqSeg.nextSegCid, newDeqSeg());
13:                 deqSeg = null;
14:                 send message(cid, false);
```

```
1: procedure newEnqSeg()
2:     enqSeg = new Segment();
3:     segQueue.enq(engSeg) ;
4:     notify CPUs of the new enqueue segment;
```

```
1: procedure newDeqSeg()
2:     deqSeg = segQueue.deq();
3:     notify CPUs of the new dequeue segment;
```

The pseudocode of the algorithm is presented in Algorithm 2. Each PIM core has local variables enqSeg and deqSeg that are references to local enqueue and dequeue segments. When enqSeg (respectively deqSeq) is not null, it indicates that the PIM core is currently holding the enqueue (respectively dequeue) segment. Each PIM core also maintains a local queue segQueue for storing local segments. CPUs and PIM cores communicate via message(cid, content) calls, where cid is the unique core ID (CID) of the receiver and the content is either a request or a response to a request.

Once a PIM core receives an enqueue request enq(cid, u) of node u from a CPU whose CID is cid, it first checks if it is holding the enqueue segment (line 2 of Procedure enq(cid, u)). If so, the PIM core enqueues u (lines 5-12), and otherwise sends back a message informing the CPU that the request is rejected (line 3) so that the CPU can resend its request to the right PIM core holding the enqueue segment (we will explain later how the CPU can find the right PIM core). After enqueuing u, the PIM core may find the enqueue segment is longer than the threshold (line 13). If so, it sends a message with a newEnqSeg() request to the PIM core of another vault that is chosen to create a new enqueue segment. Finally the PIM core sets its enqSeq to null indicating it no longer deals with enqueue operations. Note that the CID cid' of the PIM core chosen for creating the new segment is recorded in enqSeg.nextSegCid for future use in dequeue requests. As Procedure newEnqSeg()

in Algorithm 2 shows, The PIM core receiving this newEnqSeg() request creates a new enqueue segment and enqueues the segment into its segQueue (line 3). Finally it notifies CPUs of the new enqueue segment (we will get to it in more detail later).

Similarly, when a PIM core receives a dequeue request deq(cid) from a CPU with CID cid, it first checks whether it still holds the dequeue segment (line 2 of Procedure deq(cid)). If so, the PIM core dequeues a node and sends it back to the CPU (lines 5-7). Otherwise, it informs the CPU that this request has failed (line 3) and the CPU will have to resend its request to the right PIM core. If the dequeue segment is empty (line 8) and the dequeue segment is not the same as the enqueue segment (line 11), which indicates that the FIFO queue is not empty and there exists another segment, the PIM core sends a message with a newDeqSeg() request to the PIM core with CID deqSeg.nextSegCid. (We know that this PIM core must hold the next segment, according to how we create new segments in enqueue operations, as shown at lines 14-16 of Procedure enq(cid, u).) Upon receiving the newDeqSeg() request, as illustrated in Procedure newDeqSeg(), the PIM core retrieves from its segQueue the oldest segment it has created and makes it the new dequeue segment (line 2). Finally the PIM core notifies CPU that it is holding the new dequeue segment now.

Now we explain how CPUs and PIM cores coordinate to make sure that CPUs can find the right enqueue and dequeue segments, when their previous attempts have failed due to changes of those segments. We will only discuss how to deal with enqueue segments here, since the same methods can be applied to dequeue segments. A straightforward way to inform CPUs is to have the owner PIM core of the new enqueue segment send notification messages to them (line 4 of newEngSeg()) and wait until CPUs all send back acknowledgement messages. However, if there is a slow CPU that doesn't reply in time, the PIM core has to wait for it and therefore other CPUs cannot have their requests executed. A more efficient, non-blocking method is to have the PIM core start working for new requests immediately after it has sent off those notifications. A CPU does not have to reply to those notifications in this case, but if its request later fails, it needs to send messages to (sometimes all) PIM cores to ask whether a PIM core is currently in charge of the enqueue segment. In either case, the correctness of the algorithm is guaranteed: at any time, there is only one enqueue segment and only one dequeue segment, and only requests sent to them will be executed.

We would like to mention that the PIM-managed FIFO can be further optimized. For example, the PIM core holding the enqueue segment can combine multiple pending enqueue requests and store the nodes to be enqueued in an array as a "fat" node of the queue, so as to reduce memory accesses. This optimization is also used in the flat-combining FIFO queue [44]. Even without this optimization, our algorithm still performs well, as we will show next.

5.5.2 Pipelining and Performance analysis

We compare the performance of three concurrent FIFO queue algorithms—our PIM-manged FIFO queue, a flat-combining FIFO queue and a F&A-based FIFO queue [64]. The F&A-based FIFO queue is the most efficient concurrent FIFO queue we are aware of, where threads make F&A operations on two shared variables, one for enqueues and the other for dequeues, to compete for slots in the FIFO queue to enqueue and dequeue nodes (see [64] for more details). The flat-combining FIFO queue we consider is based on the one proposed by [44], with a modification that threads compete for two "combiner locks", one for enqueues and the other for dequeues. We further simplify it based on the assumption that the queue is always non-empty, so that it doesn't have to deal with synchronization issues between enqueues and dequeues when the queue is empty.

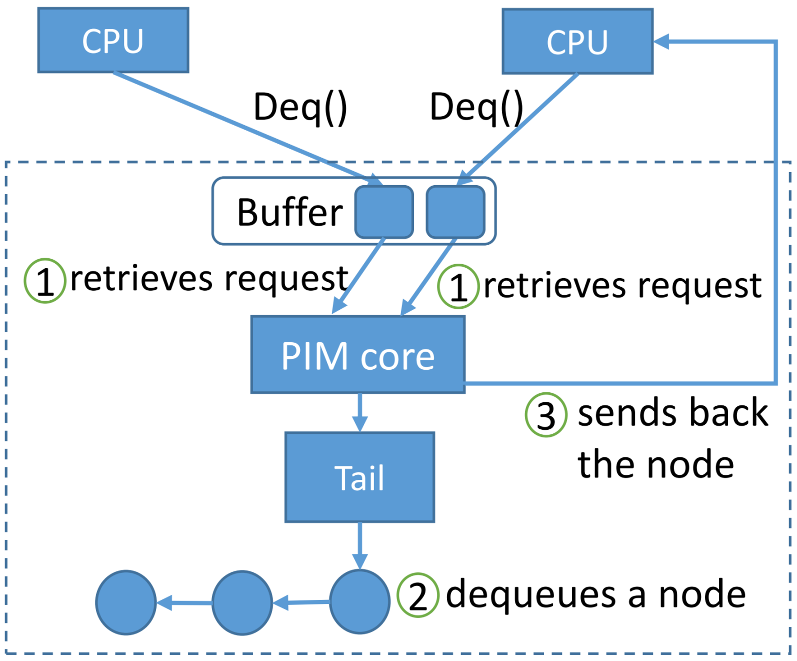
Let us first assume that a queue is long enough such that the PIM-managed FIFO queue has more than one segment, and enqueue and dequeue requests can be executed separately. Since changes of enqueue and dequeue segments happen very infrequently, its overhead is negligible and therefore ignored to simplify our analysis. (If the threshold of segment length at line 13 of enq(cid, u) is a large integer n, then, in the worst case, changing an enqueue or dequeue segment happens only once every n requests, and the cost is only the latency of sending one message and a few steps of local computation.) Since enqueues and dequeues are isolated in all the three algorithms when queues are long enough, we will focus on dequeues, and the analysis of enqueues is almost identical.

Assume there are p concurrent dequeue requests by p threads. Since each thread needs to make a F&A operation on a shared variable in the F&A-based algorithm and F&A operations on a shared variable are essentially serialized, the execution time of p requests in the algorithm is at least $p\mathcal{L}_{atomic}$. If we assume that each CPU makes a request immediately after its previous request completes, we can prove that the throughput of the algorithm is at most $\frac{1}{\mathcal{L}_{atomic}}$.

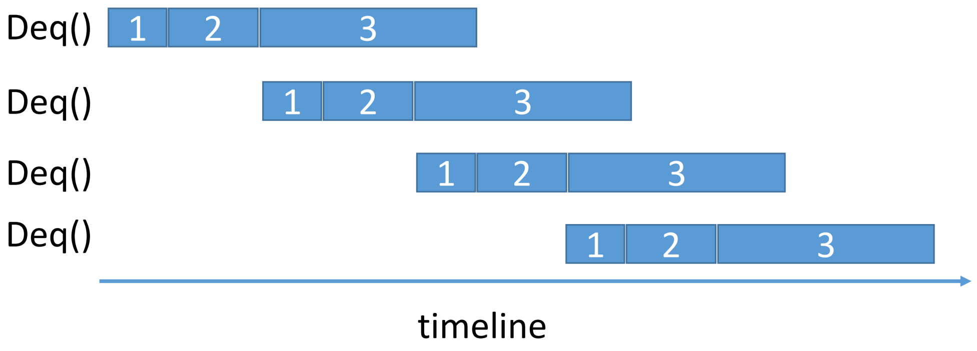
The flat-combining FIFO queue maintains a sequential FIFO queue and threads submit their requests into a publication list. The publication list consists of slots, one for each thread, to store those requests. After writing a request into the list, a thread competes with other threads for acquiring a lock to become the "combiner". The combiner then goes through the publication list to retrieve requests, executes operations for those requests and writes results back to the list, while other threads spin on their slots, waiting for the results. The combiner therefore makes two last-level cache accesses to each slot other than its own slot, one for reading the request and one for writing the result back. Thus, the execution time of p requests in the algorithm is at least $(2p-1)\mathcal{L}_{llc}$ and the throughput of the algorithm is roughly $\frac{1}{2\mathcal{L}_{llc}}$ for large enough p.

Note that we have made quite optimistic analysis for the F&A-based and flat-combining algorithms by counting only the costs in part of their executions. The latency of accessing and modifying queue nodes in the two algorithms is ignored here. For dequeues, this latency can be high: since nodes to be dequeued in a long queue is unlikely to be cached, the combiner has to make a sequence of memory accesses to dequeue them one by one. Moreover, the F&A-based algorithm may suffer performance degradation under heavy contention, because contended F&A operations may perform worse in practice.

The performance of our PIM-managed FIFO queue seems poor at first sight: although a PIM core can update the queue efficiently, it takes a lot of time for the PIM core to send results back to CPUs one by one. To improve its performance, the PIM core can *pipeline* the executions of requests, as illustrated in Figure 5.6(a). Suppose p CPUs send p dequeue requests concurrently to the PIM



(a)



(b)

Figure 5.6: (a) illustrates the pipelining optimization, where a PIM core can start executing a new deq() (step 1 of deq() for the CPU on the left), without waiting for the dequeued node of the previous deq() to return to the CPU on the right (step 3). (b) shows the timeline of pipelining four deq() requests.

core, which takes time $\mathcal{L}_{message}$. The PIM core fist retrieves a request from its message buffer (step 1 in the figure), dequeues a node (step 2) for the request, and sends the node back to the CPU (step 3). After the PIM core sends off the message containing the node, it immediately retrieves the next request, without waiting for the message to arrive at its receiver. This way, the PIM core can pipeline requests by overlapping the latency of message transfer (step 3) and the latency of memory accesses and local computations (steps 1 and 2) in multiple requests (see Figure 5.6(b)). During the execution of a dequeue, the PIM core only makes one memory access to read the node to be dequeued, and two L1 cache accesses to read and modify the tail node of the dequeue segment. It is easy to prove that the execution time of p requests, including the time CPUs send their requests to the PIM core, is only $\mathcal{L}_{message} + p(\mathcal{L}_{pim} + \epsilon) + \mathcal{L}_{message}$, where ϵ is the total latency of the PIM core making L1 cache accesses and sending off one message, which is negligible in our performance model. If each CPU makes another request immediately after it receives the result of its previous request, we can prove that the throughput of the PIM-managed FIFO queue is

$$\frac{1 - 2\mathcal{L}_{message}}{\mathcal{L}_{pim} + \epsilon} \approx \frac{1 - 2\mathcal{L}_{message}}{\mathcal{L}_{pim}} \approx \frac{1}{\mathcal{L}_{pim}},$$

which is expected twice the throughput of the flat-combining queue and three times that of the F&A queue, in our performance model assuming $\mathcal{L}_{atomic} = 3\mathcal{L}_{llc} = 3\mathcal{L}_{pim}$.

When the PIM-managed FIFO queue is short, it may contain only one segment which deals with both enqueue and dequeue requests. In this case, its throughput is only half of the throughput shown above, but it should still be at least as good as the throughput of the other two algorithms.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA '00, pages 1–12, New York, NY, USA, 2000. ACM.

[2] Umut A. Acar, Arthur Chargueraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 219–228, New York, NY, USA, 2013. ACM.

[3] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive work stealing with parallelism feedback. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 112–120, New York, NY, USA, 2007. ACM.

[4] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 105–117, New York, NY, USA, 2015. ACM.

[5] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 336–348, New York, NY, USA, 2015. ACM.

[6] Berkin Akin, Franz Franchetti, and James C. Hoe. Data reorganization in memory using 3d-stacked dram. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 131–143, New York, NY, USA, 2015. ACM.

[7] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[8] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, October 1989.

[9] Erfan Azarkhish, Christoph Pfister, Davide Rossi, Igor Loi, and Luca Benini. Logic-base interconnect design for near memory computing in the smart memory cube. *IEEE Trans. VLSI Syst.*, 25(1):210–223, 2017.

[10] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. High performance axi-4.0 based interconnect for extensible smart memory cubes. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 1317–1322, San Jose, CA, USA, 2015. EDA Consortium.

[11] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In *Proceedings of the 29th International Conference on Architecture of Computing Systems – ARCS 2016 - Volume 9637*, pages 19–31, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[12] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. Die stacking (3d) microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 469–479, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, March 1996.

[14] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366, New York, NY, USA, 2011. ACM.

[15] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 1–12, New York, NY, USA, 1995. ACM.

[16] Guy E. Blelloch and Margaret Reid-Miller. Pipelining with futures. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 249–259, New York, NY, USA, 1997. ACM.

[17] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, SPAA '96, pages 297–308, New York, NY, USA, 1996. ACM.

[18] 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 *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[19] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27(1):202–229, February 1998.

[20] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[21] Amirali Boroumand, Saugata Ghose, Brandon Lucia, Kevin Hsieh, Krishna Malladi, Hongzhong Zheng, and Onur Mutlu. Lazypim: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters*, 2016.

[22] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM.

[23] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[24] Quan Chen, Minyi Guo, and Zhiyi Huang. Cats: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 163–172, New York, NY, USA, 2012. ACM.

[25] Rezaul Alam Chowdhury, Vijaya Ramachandran, Francesco Silvestri, and Brandon Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, 73(7):911–925, 2013.

[26] Hybrid Memory Cube Consortium. Hybrid memory cube specification 1.0.

[27] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[28] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.

[29] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P Sadayappan, and Chau-Wen Tseng. Dynamic load balancing of unbalanced computations using message passing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[30] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM.

[31] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 209–220, New York, NY, USA, 1995. ACM.

[32] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in manticore. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 119–130, New York, NY, USA, 2008. ACM.

[33] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[34] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[35] John Giacomoni, Tipp Moseley, and Manish Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 43–52, New York, NY, USA, 2008. ACM.

[36] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, April 1995.

[37] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, pages 64–75, New York, NY, USA, 1989. ACM.

[38] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 151–162, New York, NY, USA, 2006. ACM.

[39] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst.*, 5(2):164–189, April 1983.

[40] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, William Athas, Vincent Freeh, Jaewook

Shin, and Joonseok Park. Mapping irregular applications to diva, a pim-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, New York, NY, USA, 1999. ACM.

[41] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 9–17, New York, NY, USA, 1984. ACM.

[42] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

[43] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, OPODIS'05, pages 3–16, Berlin, Heidelberg, 2006. Springer-Verlag.

[44] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[45] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.

[46] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Scalable concurrent counting. *ACM Trans. Comput. Syst.*, 13(4):343–364, November 1995.

[47] Maurice Herlihy and Zhiyu Liu. Well-structured futures and cache locality. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 155–166, New York, NY, USA, 2014. ACM.

[48] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[49] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[50] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *IEEE 34th International Conference on Computer Design, ICCD 2016*, pages 25–32. IEEE, 2016.

[51] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new dram architecture increases density and performance. In *Symposium on VLSI Technology, VLSIT 2012*, pages 87–88. IEEE, 2012.

[52] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Vi Lam, Josep Torrellas, and Pratap Pattnaik. Flexram: Toward an advanced intelligent memory system. In *Proceedings of the IEEE International Conference On Computer Design*, ICCD '99.

[53] Alex Kogan and Maurice Herlihy. The future(s) of shared data structures. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 30–39, New York, NY, USA, 2014. ACM.

[54] Peter M. Kogge. Execube-a new architecture for scaleable mpps. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01*, ICPP '94, pages 77–84, Washington, DC, USA, 1994. IEEE Computer Society.

[55] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-t: a high-performance parallel lisp. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 81–90, New York, NY, USA, 1989. ACM.

[56] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. On-the-fly pipeline parallelism. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, SPAA '13, pages 140–151, New York, NY, USA, 2013. ACM.

[57] Joao V. F. Lima and Nicolas Maillard. Online mapping of mpi-2 dynamic tasks to processes and threads. *Int. J. High Perform. Syst. Archit.*, 2(2):81–89, March 2009.

[58] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 260–267, New York, NY, USA, 1988. ACM.

[59] Gabriel H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 453–464, Washington, DC, USA, 2008. IEEE Computer Society.

[60] Hans-Wolfgang Loidl and Kevin Hammond. On the granularity of divide-and-conquer parallelism. In *Proceedings of the 1995 International Conference on Functional Programming*, FP'95, pages 135–144, Swinton, UK, UK, 1995. British Computer Society.

[61] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Gener. Comput. Syst.*, 30:202–215, January 2014.

[62] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS '11, 2011.

[63] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the Seventeenth Annual ACM Symposium*

on Parallelism in Algorithms and Architectures, SPAA '05, pages 253–262, New York, NY, USA, 2005. ACM.

[64] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.

[65] Stephen Olivier and Jan Prins. Scalable dynamic load balancing using upc. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 123–131. IEEE, 2008.

[66] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. Openmp task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl.*, 26(2):110–124, May 2012.

[67] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 192–203, Washington, DC, USA, 1998. IEEE Computer Society.

[68] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, March 1997.

[69] W. Pugh. Concurrent maintenance of skip lists. Technical report, University of Maryland at College Park, 1990.

[70] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, EuroPar'10, pages 217–229, Berlin, Heidelberg, 2010. Springer-Verlag.

[71] Kaushik Ravichandran, Sangho Lee, and Santosh Pande. Work stealing for multi-core hpc clusters. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 205–217, Berlin, Heidelberg, 2011. Springer-Verlag.

[72] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks: Preliminary version. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 54–63, New York, NY, USA, 1995. ACM.

[73] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 91–100, New York, NY, USA, 2009. ACM.

[74] Harold S. Stone. A logic-in-memory computer. *IEEE Trans. Comput.*, 19(1):73–78, January 1970.

[75] J. Valois. *Lock-free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 1996.

[76] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. *SIGPLAN Not.*, 36(7):34–43, June 2001.

[77] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 439–453, New York, NY, USA, 2005. ACM.

[78] Yonghong Yan, Sanjay Chatterjee, Zoran Budimlic, and Vivek Sarkar. Integrating mpi with asynchronous task parallelism. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 333–336, Berlin, Heidelberg, 2011. Springer-Verlag.

[79] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 85–98, New York, NY, USA, 2014. ACM.

[80] Qiuling Zhu, Berkin Akin, H. Ekin Sumbul, Fazle Sadi, James C. Hoe, Larry T. Pileggi, and Franz Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *IEEE International 3D Systems Integration Conference, 3DIC 2013, San Francisco, CA, USA, October 2-4, 2013*, pages 1–7, 2013.

[81] Qiuling Zhu, Tobias Graf, H. Ekin Sumbul, Larry T. Pileggi, and Franz Franchetti. Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware. In *IEEE High Performance Extreme Computing Conference, HPEC 2013, Waltham, MA, USA, September 10-12, 2013*, pages 1–6, 2013.