15. Hierarchical Models and Software Tools for Parallel Programming

Massimo Coppola and Martin Schmollinger

15.1 Introduction

Hierarchically structured architectures are becoming more and more pervasive in the field of parallel and high performance computing. While memory hierarchies have been recognized for a long-time, only in the last years hierarchical parallel structures have gained importance, mainly as a result of the trend towards cluster architectures and high-performance application of computational grids.

The similarity among the issues of managing memory hierarchies and those of parallel computation has been pointed out before (see for instance [213]). It is an open question if a single, unified model of both aspects exists, and if it is theoretically tractable. Correspondingly, a programming environment which includes support for both hierarchies is still lacking. We thus need well-founded models and efficient new tools for hierarchical parallel machines, in order to connect algorithm design and complexity results to high-performance program implementation.

In this chapter we survey theoretically relevant results, and we compare them with existing software tools and programming models. One aim of the survey is to show that there are promising results with respect to the theoretical computational models, developed by merging the concepts of bulk-parallel computational models with those from the hierarchical memory field. A second goal is to investigate if software support has been realized, and what is still missing, in order to exploit the full performance of modern high-performance cluster architectures. Even in this case, solutions emerge from combining results of different nature, those employing hardware-provided shared memory and those explicitly dealing with message passing. We will see that both at the theoretical level and on the application side, combination of techniques from different fields is often promising, but still leaves many open questions and unresolved issues.

The chapter is organized in three parts. The first one (Sect. 15.2) describes the architectural background of current parallel platforms and supercomputers. The basic architectural options of parallel architectures are explained, showing that they naturally lead to a hierarchy concept associated with the exploitation of parallelism. We discuss the technological reasons for, and future expectations of current architectural trends.

The second part of the chapter gives an overview of parallel computational models, exploring the connection among the so-called *parallel bridging models*

U. Meyer et al. (Eds.): Algorithms for Memory Hierarchies, LNCS 2625, pp. 320-354, 2003.

and external memory models, here mainly represented by the *parallel disk model* (PDM) [754]. There is a similarity between problems in bulk parallelism and block-oriented I/O. Both techniques try to efficiently exploit locality in mapping algorithmic patterns to a hierarchical structure. We discuss the issues of parallel computation models in Sect. 15.3. In Sect. 15.4 we get to discuss parallel bridging models. We survey definitions and present some models of the class. We describe their extensions to hierarchical parallelism, and survey results on emulating their algorithms using sequential and parallel external-memory models. At the end of Sect. 15.4 we describe two results that exploit parallel hierarchical models for algorithm design.

The third part of the chapter shifts toward the practical approach to hierarchical architectures. Sect. 15.5 gives an overview of software tools and programming models that can be used for program implementation. We consider libraries for parallel and external-memory programming, and combined approaches. With respect to parallel software tools, we focus on the existing approaches which support hierarchy-aware program development. Section 15.6 summarizes the chapter and draws conclusions.

15.2 Architectural Background

In the following, we assume the reader is familiar with the basic concepts of sequential computational architectures. We also assume the notions of process and thread[1] are known. Table 15.1 summarizes some acronyms used in the chapter.

Parallel architectures are made up from multiple processing and memory units. A network connects the processing units and the memory banks (see Fig. 15.1a). We refer the reader to [484], which is a good starting point to understand the different design options available (the kind of network, which modules are directly connected, etc.). We only sketch them here due to lack of space.

Efficiency of communication is measured by two parameters, *latency* and *bandwidth*. Latency is the time taken for a communication to complete, and bandwidth is the rate at which data can be communicated. In a simple world, these metrics are directly related. For communication over a network, however, we must take into account several factors like physical limitations, communication startup and clean-up times, and the possible performance penalty from many simultaneous communications through the network. As a very general rule, latency depends on the network geometry and implementation, and bandwidth increases with the length of the message, because of the lesser

[1] Largely simplifying, a *process* is a running program with a set of resources, which includes a private memory space; a *thread* is an activity within a process. A thread has its own control flow but shares resources and memory space with other threads in the same process.

and lesser influence of fixed overheads. Communication latency and bandwidth dictate the best *computational grain* of a parallel program. The grain is the size of the (smallest) subproblem that can be assigned to a different processor, and plays a main role in the trade-off between exploited parallelism and communication overheads.

Extreme but realistic examples of networks are the *bus connection* and the $n \times n$ *crossbar*. The former is a common hardware channel which can link only a pair of modules at a time. It has the least circuital complexity and cost. The latter is a square matrix of switches that can connect up to n non-conflicting pairs of modules. It achieves the highest connectivity at the highest circuital cost. A lot of systems use structures that are in between these two, as a compromise between practical scalability and performance.

Different memory organizations and caching solutions have been devised to improve the memory access performance (see Fig. 1b, 1c).

Almost all modern parallel computers belong to the MIMD class of parallel architectures. This means that processing nodes can execute indepent programs over possibly different data. The MIMD class is subdivided in [744] according to the characteristic of the physical memory, into *shared memory* MIMD (SM-MIMD) and *distributed memory* MIMD (DM-MIMD).

The memory banks of a shared memory MIMD machine form a common address space, which is actively supported by the network hardware. Different processors can interfere with each other when accessing the same memory module, and race conditions may show up in the behaviour of the programs. Therefore, hardware lock and update protocols have to be used to avoid inconsistencies in memory and among the caches[2]. Choosing the right network structure and protocols are critical design issues, which drive the performance of the memory system. Larger and larger shared memory machines lead to difficult performance problems.

Multi-stage crossbars (networks made of smaller interconnected crossbars) are getting more and more important with the increasing number of processors in shared memory machines. If the processors are connected by a multi-stage crossbar and each processor has some local memory banks, there is a memory hierarchy within the shared memory of the system.

[2] We do not analyze in depth here either cache coherence issues (see Chapter 16), multi-stage networks or cache-only architectures [484].

Table 15.1. Main acronyms used throughout the chapter.

MIMD	multiple instruction, multiple data	EM	external memory
SM-MIMD	shared-memory MIMD	PDM	parallel disk model
DM-MIMD	distributed-memory MIMD	HMM	hierarchical memory model
SMP	symmetric multiprocessor	PBM	parallel bridging model
		BSP	see Sec. 15.4, page 329
SDSM	software distributed shared memory	CGM	see Sec. 15.4, page 330
LAN	local area network	QSM	see Sec. 15.4, page 331
		LogP	see Sec. 15.4, page 331

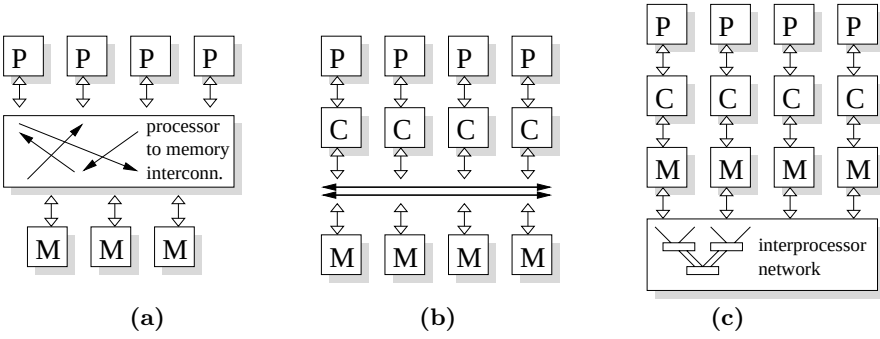

Fig. 15.1. Overall structure of DM and SM-MIMD architectures. (a) Generic MIMD machines have multiple processors and memory banks (not necessarily the same number) – (b) example of SMP with multiple memory banks, processor caches and a bus interconnection – (c) example of a MIMD architecture composed of single-processor nodes and a more sophisticated interconnection network. Depending on the network implementation, this can be either a DM-MIMD or a NUMA SM-MIMD architecture.

Systems in which the access from a processor to some of the memory banks, e.g. its local one, is faster than access to the rest of the memory are called non-uniform memory access systems (NUMA), in contrast with uniform memory access systems (UMA). Shared memory architectures, both uniform and non-uniform ones, are often called Symmetric Multiprocessors (SMP), because the architecture is fully symmetric from the point of view of the running programs.

In contrast to the shared memory machines, in a distributed memory machine each processing node has its own address space. Therefore, it is up to the user to define efficient data decompositions and explicit data exchange patterns for the applications. Even distributed memory machines have to confront with issues concerning the interconnection structure among the processing nodes. Popular networks for distributed memory machines are the hypercube, the 2D or 3D meshes, and multi-stage crossbars. Each processing node has its own local memory, so distributed memory architectures obviously are non-uniform memory access architectures. The network is inherently slower than the local memory, hence we have a memory hierarchy in distributed memory MIMD machines too. However, distributed memory architectures are less demanding with respect to the interconnection network[3], so they are much more scalable than the shared memory ones.

In recent years, a strong trend has emerged in the field of high performance computers towards two kinds of architectures, (1) clusters of vector computers and (2) clusters of scalar uni- and multiprocessors. Looking at the list of the

[3] For instance, coherence and locking problems are not dealt with at the hardware level. This removes some design constraints, and reduces communication overheads.

fastest 500 supercomputers in the world, the majority of them belongs to these two classes [103, 549], with the latter steadily gaining more share.

Especially clusters of SMP nodes (*SMP clusters*, in the following) are a more and more popular architecture for building supercomputers. At different scales, these systems can be classified both as distributed memory and as shared memory architectures, because SM-MIMD processing nodes are connected together to form a larger distributed memory machine. The result is a powerful parallel architecture, which combines the high effectiveness of small shared-memory computing nodes with the scalability of the distributed memory parallelism among the nodes.

In principle, we could classify SMP clusters either as shared memory or as distributed memory MIMD depending on the existence of a common address space abstraction for all the processors. A shared space can be provided by the network hardware and firmware, or only by software means on top of a general purpose network (see Section 15.5.3). However, in the latter case algorithms which exploit memory locality within SMP nodes incur much lesser communication overheads. We can thus enhance performance if we explicitly consider SMP clusters as architectures with a *parallel hierarchy* of at least two levels. The number of levels may actually be higher, depending on the topology of the intra- and inter-node networks.

Grid or metacomputing technologies [309], where supercomputers or clusters of workstations are connected with each other to run applications, result in even more levels and a less regular parallel hierarchy. Broadband connec-

Table 15.2. Parameters for different levels in a hierarchical parallel architecture. The reference commodity architecture is an Intel IA32 microprocessor core from a Pentium4/Xeon. Network bandwidth and latency are measured w.r.t. a node.

Layer	Peak Bandwidth	Latency Best	Latency Worst	Notes
CPU and caches	10-100 GB/s	$<$ 1 ns	200 ns	(1)
SM communication	$<$ 1 GB/s	200 ns	$\sim$1000 ns	(2)
DM communication	1-400 MB/s	$\sim$2 μs	2 ms	(3,4)
Local I/O	10-100 MB/s	4 ms	30 ms	(4)
DM communication (WAN)	10-100 MB/s	2 ms	1s	(5)

1. Worst-case latency is that of reading a block from memory into the external cache. In the IA32 architecture, a cache line is 64 bytes.
2. Processors share the same memory bus through separate external caches. A shared memory communication implies at least a L2 cache fault. The worst-case accounts for other issues like acquiring hardware and software locks, and thread scheduling.
3. From slow Ethernet up to Gbit Ethernet and Myrinet [744].
4. We do not include here using in parallel several disks, and multiple network interfaces per node.
5. Multi-Gigabit geographic networks are being already built, but wide area network (WAN) have higher latencies [309, sec. 21.4].

tions, ranging from local area networks to geographic ones, add more levels to the hierarchy, with different communication bandwidth and latency [309, chapter 2].

Summing up, in modern parallel architectures we have the following hierarchy of memory and communication layers.

- shared memory
- distributed memory
- local area network
- wide area network

Each one of these layers may exhibit hierarchical effects, depending on its implementation choices.

The effects on latency and bandwidth of the parallel hierarchy are similar and combine with those of the ordinary memory hierarchy. A crucial observation is that there is no strict order among the levels of these two hierarchies, which we can easily exploit to build a unitary model. For instance, we can see in Table 15.2 that the communication layers (both shared memory and distributed memory based ones) provide a bandwidth lower than main memory, and in some cases lower than that of local I/O. However, their latency is usually much lower than that of mechanical devices like disks. Different access patterns thus lead to different relative performances of communication and I/O.

Assessing the present and future characteristics of the parallel hierarchy [193] and devising appropriate programming models to exploit it are among the main open issues in modern parallel/distributed computing research.

15.2.1 Motivation and Technological Perspective

As we explained in the last section, parallel computing architectures employ memory and parallel hierarchies. In the following we summarize arguments that explain the trend towards even more hierarchical architectures than SMP clusters, and we discuss possible future developments.

In [438] some main advantages of SMP cluster architectures are found, most of them being technological and economical considerations.

- Standard off-the-shelf processors are getting faster and faster, even with respect to special purpose architectures. Because of mass production, the performance/price ratio of commercial parts is consistently better than that of special purpose processors (e.g. vector processors). Architectures made from commodity processors are going to be increasingly preferred to build SMPs, massively parallel and cluster machines.
- A similar effect shows up for network and other architecture components. Clusters ans small SMP, which employ standard components, will take advantage of this phenomenon and will become cheaper and more scalable. As soon as the performance advantages of the special purpose networks

for massive parallelism disappear, SMP clusters will supersede massively parallel architectures.
- SMP clusters are scalable and expandable. Their architecture is intrinsically more scalable, and it is moreover practically expandable by adding more nodes and/or upgrading processing nodes. While it is often not possible to add processors in a monolithic SMP or in a special purpose connection, it is easy to build a SMP cluster step by step.
- The size of memory, disk subsystem capacity and bandwidth are critical resources in a supercomputer. SMP clusters are characterized by a greater total memory size and number of disks. Hence, it is possible to have more active jobs, which even have larger data storage available.
- Still according to [438], most software for massively parallel processors or SMPs can easily be ported to SMP clusters achieving similar efficiency. With the knowledge of software for SMP machines and the already existing software for the massively parallel processors, it should be possible to provide a powerful environment for parallel software development and execution. We will give an overview of the efforts in this direction in Sect. 15.5.

Some of the preceding considerations have been recognized years ago, while others are a more recent discovery. According to Bell and Gray [103], a similar trend will last for more than a while. They depict a scenario of the evolution of parallel computers and computing grid architectures, which implies a change in the role played by Beowulf architectures and supercomputing centers.

Beowulf clusters are parallel machines built using commodity hardware and software, where the hardware can even be a mixture of uniprocessor workstations and small SMPs. Companies and research institutes, formerly users of supercomputers and proprietary software, will gradually start to build their own large Beowulf machines, which are more cost-effective than remotely hosted super-computers. Moreover, thanks also to the increased Internet bandwidth, cluster and Grid [309] technologies will merge. It will then become possible to merge Beowulf computational resources into larger, geographically distributed clusters. Applications that tolerate high communication latencies will easily be able to exploit a processing power measured in Teraflops.

In this perspective, applications which need a large amount of shared memory are seen as a weak point of Beowulf clusters. Computing centers will still exist and host large vector machines and exotic architectures (like processor-in-memory and cellular supercomputers) for the sake of running these applications. In addition, centers will have the role of resource brokers for computational grids, managing net-distributed clusters, and will provide storage for peta-scale data sets. According to this analysis [103], hierarchical parallel systems will be the principal computing structure in the future. Therefore, research on programming environments and in understanding hierarchical parallelism has an essential role.

15.3 Parallel Computational Models

We have already seen in this book that the classical random access machine (RAM) sequential model, the archetype of the von Neumann computer, does not properly account with the cost of memory access within a hierarchy of memories. The PDM model [754], and more complex multi-level computational models have been developed to increase prediction accuracy with respect to the practical performance of algorithms.

The same has happened in the field of parallel algorithms. The classical parallel random access machine (PRAM) computational model is made up of a number of sequential RAM machines, each one with its local memory. These "abstract processors" compute in parallel and can communicate by reading and writing to a global memory which they all share. Several precise assumptions are made to keep the model general.

- Unlimited resources: no bound is put on the number of processors, the size of local or global memory.
- Parallel execution is fully synchronous, all active processor always complete one instruction in one time unit.
- Unitary cost of memory access, both for the local and global memory.
- Unlimited amount of simultaneous operations in the global memory is allowed (though same-location collisions are forbidden).

These assumptions are appropriate for a theoretical model. They allow to disregard the peculiarities of any specific architecture, and make the PRAM an effective model in studying abstract computational complexity. However, they are not realistic for the majority of physical architectures, as practical bandwidth constraints, network traffic constraints and locality effects are completely ignored.

Considering the issues discussed in Sect. 15.2, we see that real MIMD machines are much more complex. Synchronous parallel execution is usually impossible on modern parallel computers, as well as to ensure constant, uniform memory access times independently of machine size, amount of exchanged data and exploited parallelism. Indeed, optimal PRAM complexity is often misguiding with respect to real computational costs.

Several variants of the PRAM model have been devised with the aim of reconciling theoretical computational costs with real performance. They add different kinds of constraints and costs on the basic operations. A survey on these derived models is given in [354]. We do not even discuss the research on communication and algorithmic performance of models which use a fixed network structure (e.g. a mesh or hypercube). Despite the results on network cross-simulation properties, network-specific algorithms are often too tied to the geometry of the network, and show a sub-optimal behavior on other kinds of interconnection.

We focus on different research track, which has started in the last years and involves the class of parallel bridging computational models. Parallel

bridging models aim at exploiting a higher degree of hardware independence, and some of them explicitly consider the grain of the computation as a key factor in devising efficient and practical algorithms. In the following section we concentrate on this approach to modeling performance. We start from a description of the approach and of the most widely adopted models of this class, and then we survey some of the results about the connections with hierarchical memory models for sequential programming.

15.4 Parallel Bridging Models

Computational models should exhibit the right balance of abstraction and detail, to reflect the actual behavior of parallel algorithms while keeping the analysis tractable. As in the sequential case, ideal parallel models should satisfy both efficiency and universality requirements, so that most results about computational costs can be applied in practice and that they are essentially unaffected by the underlying architecture.

The need to develop such a "reference" parallel model has lead in the 1990s to the concept of a *parallel bridging model* (PBM). A bridging model should act as a standard, aiming at the best separation of algorithm and software development from the architecture. Several models of this class have been developed. They use a more abstract approach in modeling the interconnection architecture.

One of the pioneering works in the field is [742] by Valiant, where the following goals stemming from the initial definition are stated.

Cost measure. A PBM should have a cost measure to guide algorithm development, detailed enough for accurate performance prediction. The model should be independent of a specific architecture and technology, but it should reflect the fundamental constraints of parallel machines.

Efficient universality. Mapping high-level PBM algorithms onto actual machines should not lead to reduced efficiency. We wish to avoid logarithmic simulation losses, and aim at constant bounded inefficiency, so that we can afford developing algorithms for the PBM model only.

Neutrality and Portability. A PBM should be neutral with respect to the number of processors, i.e. results should be applicable not only asymptotically, but also for very small parallel machines. It should also allow the programmer to write fully portable programs, avoiding explicit memory management, low-level communications and synchronizations. We are thus requiring that our computational model is a good *programming* model too.

Parallel slackness. This is the amount of excess parallelism in the algorithm needed to achieve optimal execution. A PBM program written for v virtual processors should be optimally simulated on p physical processors if p is rather smaller than v (e.g. $v = p \log p$). The approach ensures that there is a

p	number of processors
L	message latency / synchronization
g	cost parameter for message routing

for processor i in superstep t

$w_{i,t}$	local computation
$\lambda_{i,t}$	num. sent messages
$\mu_{i,t}$	num. received messages
$w_t = \max_i w_{i,t}$	global work in t
$h_t = \max_i \max\{\lambda_{i,t}, \mu_{i,t}\}$	global routing in t
$w_t + g \cdot h_t + L$	cost of superstep t

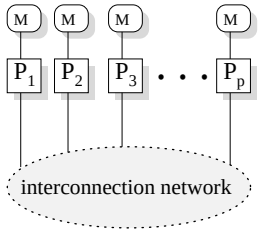

Fig. 15.2. BSP symbols and parameters (*left*). The BSP abstract architecture (*right*).

chance to superimpose computation and communication of different virtual processors on a wide range of interconnection networks. The higher degree of asynchronism introduced in PBMs helps to avoid that fine-grain parallelism negatively affects the algorithm execution.

15.4.1 The Bulk Synchronous Parallel Model

The bulk synchronous parallel model (BSP) that Valiant proposes (as described in [75]) is made up of a set of p processing nodes with local memories and a complete interconnection network which delivers messages between pairs of nodes (Fig. 15.2). Three parameters are used to specify the underlying hardware: the number p of processors, a latency parameter L, which is the maximum latency of a message or synchronization in the network, and a gap parameter g, which is the amount of time that a processor must wait after a send operation before sending a new message.

A BSP computation consists of *supersteps*. During a superstep processors compute using their local memory and exchange a certain amount of messages with each other. Messages sent during superstep t are received only at the beginning of superstep $t + 1$. The components of a superstep, represented in Fig. 15.3a, are thus a computation phase (the grey stripes), a varying routing relation (which expresses the pattern of message exchange) and the constant-bounded synchronization time L (the white vertical stripes). The table of Fig. 15.2 summarizes the essential notation of the BSP model, with respect to a superstep t and a processing node i.

We define $h_{i,t} = \max(\lambda_{i,t}, \mu_{i,t})$ as the largest number of messages sent or received by processor i during current superstep. The routing relation among the nodes (the relation among sender and receiver processors for the set of all messages) has size $h_t = \max_i h_{i,t}$ for superstep t. It is usually called h-relation, to emphasize the fact that we look at its h parameter.

With these parameter definitions, w_t and h_t are the largest w and h values in superstep t. If we imagine that supersteps are globally synchronized

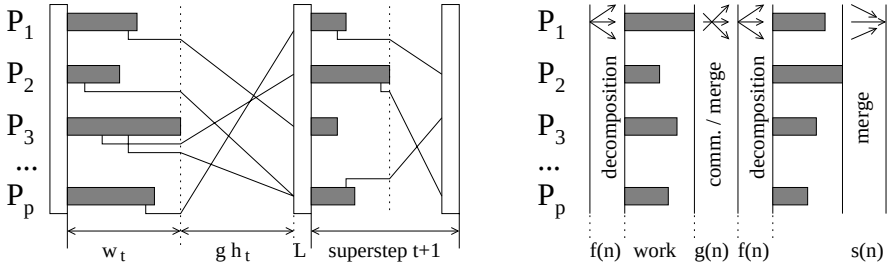

Fig. 15.3. (a) BSP superstep execution (b) CGM supersteps

by barriers (i.e. all units must complete a superstep before any of them can proceed to the next superstep), we can estimate the length of a superstep in time units as $w_t + g \cdot h_t + L$. This value becomes an upper bound if we assume instead that synchronization is only enforced when actually needed (e.g. before receiving a message).

We can analyze a BSP algorithm by computing w_t and h_t for each superstep. If the algorithm terminates in T supersteps, the *local work* $W = \sum_t w_t$ and the *communication volume* $H = \sum_t h_t$ of the algorithm lead to the cost estimate $W + g \cdot H + L \cdot T$. A more sound evaluation compares the performance of the algorithm with that of the best known sequential algorithm. Let T_{seq} be the sequential running time, we call *c-optimal* a BSP algorithm that solves the problem with $W = c \cdot T_{\text{seq}}/p$ and $g \cdot H + L \cdot T = o(T_{\text{seq}}/p)$ for a constant c.

Other parallel bridging models have been developed. Among them we mention the CGM model, which is the closest to the BSP, the LogP and the QSM models, which we describe below.

The Coarse-Grained Multicomputer. The *coarse-grained multicomputer* (CGM) model [245] is based on supersteps too. The communication phase in CGM is different from that of BSP, as it involves all processors in a global communication pattern, and $O(n)$ data are exchanged at each communication phase. In Fig. 15.3b the different patterns are the f, g, s functions. Only two numeric parameters are used, the number of nodes p and the problem size n. Each node has thus $O(n/p)$ local memory.

In the original presentation the model was parametric, as the network structure was left essentially unspecified. The communication phases were allowed to be any global pattern (e.g. sorting, broadcasts, partial sums) which could be efficiently emulated on various interconnection networks. To get the actual algorithmic cost, one should substitute the routing complexity of the parallel patterns on a given network (e.g. $g(n, p)$ may be the complexity of exchanging $O(n/p)$ keys in a hypercube of diameter $\log_2 p$).

The challenge in the CGM model is to devise a coarse-grain decomposition of the problem into independent subproblems by exploiting a set of "portable" global parallel routines. The best algorithms will usually require the smallest

possible number of supersteps. During the years, in the common use CGM has been simplified and became close to BSP. In recent works [243, 244], the network geometry is no longer considered. CGM algorithms are defined as a special class of BSP algorithms, with the distinguishing feature that each CGM superstep employs a routing relation of size $h = \Theta(n/v)$.

LogP Model. In the LogP [233] model, processors communicate through point-to-point messages, ignoring the network geometry like in BSP. Unlike BSP and the other PBMs, there are no supersteps in LogP.

LogP models physical communication behavior. It uses four parameters: l, an upper bound on communication latency; o, the overhead involved in a communication; g, a time gap between sending two messages; and the physical parallelism P. Messages are considered to be of small, fixed length, thus introducing the need to split large communications. There are two flavors of the model, *stalling* LogP, which imposes a network capacity constraint (a processor can have no more than $\lceil l/g \rceil$ messages in transit to it at the same time, or senders will stall), and *non-stalling* LogP, which has no constraint.

Because of the unstructured and asynchronous programming model, of the need to split messages into packets, and to deal with the capacity constraint, algorithm design and analysis with LogP is more complex than with the other PBMs. There are comparably fewer results with LogP, even if most basic algorithms (broadcasts, summing) have been analyzed.

QSM Model. The *queuing shared memory* (QSM) [330] model can be seen both as a PRAM evolution and as a shared-memory variant of the BSP. Like a PRAM, a set of processors with private memories communicate by means of a shared memory. Like in the BSP, QSM computation is globally divided into phases. Read and write operation are posted to the shared memory, and they complete at the end of a phase. Concurrent reads or writes (but not both) to a memory location are allowed.

Each processor must also perform a certain amount of local computation within each phase. The cost of each phase is defined as $\max(m_{op}, g \cdot m_{rw}, \kappa)$, where m_{op} is the largest amount of local computation in the phase, m_{rw} is the largest number of shared reads and writes from the same processor, and the gap parameter g is the overhead of each request. Latency is not explicitly considered, and it is substituted by the *maximum contention* κ of the phase, i.e. the maximum number of colliding accesses on any location in that phase. A large number of algorithms designed for variants of the PRAM can be easily mapped on the QSM.

A Comparison of Parallel Bridging Models. Several results about emulation among different parallel bridging models can be found in the literature. Emulations are *work-preserving* if the product $p \cdot t$ (processors per execution time) on the emulating machine is the same as that on the machine being emulated, to within a constant factor. Work-preserving emulations typically increase the amount of parallel slackness (the emulating machine has fewer

processor than the emulated one), and are characterized by a certain *slowdown*. The slowdown is $O(f)$, when we are able to map an algorithm, running in t time on p processors, to one running on $p' \leq p/f$ processors in time $t' = O(t \cdot (p/p'))$. An ideal slowdown of 1 means that the emulation introduces at most a constant factor of inefficiency. Table 15.3 summarizes some asymptotic slowdown results taken from a recent survey by Ramachandran [622]. The fact that a collection of work-preserving emulations with small slowdown exists, suggests that these models are to a good extent equivalent in their applicability as cost models to real parallel machines.

However, some of the bridging models are better suited than others for the role of programming models, as a more abstract view of the algorithm structure and communication pattern allows easier algorithm design and analysis.

From this point of view, LogP is probably the hardest PBM to use. It leads to difficult, low-level analysis of communication behavior, and thus it has been rarely used to evaluate complex algorithms. The QSM can be used to evaluate the practical performance of many existing PRAM algorithms, but is a low-level model too. QSM is a "flat" model, which disregards the hierarchical structure of the computation, and it has an abstract but fine-grain approach to communication cost.

The bulk parallel models (BSP and CGM) have been used more extensively to code parallel algorithms. They proved to be easier to use when designing algorithms, and actually several software tools have been designed to directly implement BSP algorithms. In the same direction there are even more simplified model, like the one used in [690]. Aggregate computation cost is measured in terms of total work, total network traffic (sum of messages) and total number of messages. Thus the three "weights" of these operations are the parameters of this model. It can be seen as a flat, close relative of BSP and CGM, and at least a class of algorithms based on computation phases with limited unbalancing can be analyzed using this model.

Both the CGM, and extensions of the original BSP model, allow to represent hierarchically structured networks. Finally, the concepts of parallel slackness, medium-grain parallelism and supersteps have been exploited to develop efficient emulation of BSP and CGM algorithms in external memory,

Table 15.3. Slowdown of work-preserving emulation between PBMs. Most of these results concern randomized emulation algorithms. See [622] for details and references.

	Emulating model		
Emulated model	BSP	LogP (stalling)	QSM
BSP		$\log^4 p + (L/g)\log^2 p$	$\lceil (g \log p)/L \rceil$
LogP (non stalling)	L/l (det.)	1 (det.)	$\lceil (g \log p)/l \rceil$
QSM	$(L/g) + g \log p$	$\log^4 p + (l/g)\log^2 p + g \cdot \log p$	

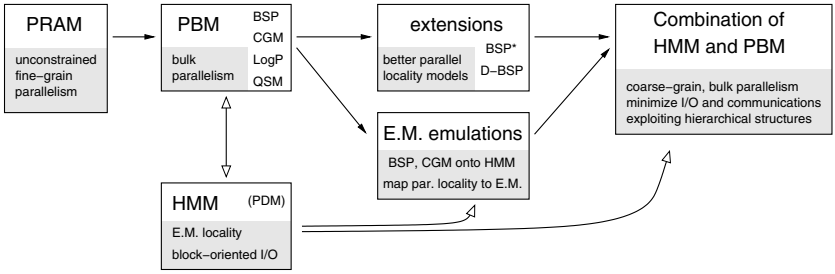

Fig. 15.4. Development path of parallel bridging models and their relationship with hierarchical memory models

showing up the connections among the design of parallel algorithms and that of external-memory algorithms.

15.4.2 Parallel Bridging Models and Hierarchical Parallelism

As earlier noted by Cormen and Goodrich [213], the bulk-processing nature of external memory algorithms is close to the bulk-parallel approach of most parallel bridging models. Both kinds of models aim at properly accounting in an abstract way for a kind of access locality. Moreover, most practical problems involving large-scale data structures are definitely a target both for parallel computing and for secondary memory computing techniques. Cormen and Goodrich fostered the development of a single computation model which included bulk-like measures for computation, communication and I/O. This will be our main topic in the following.

Two different research paths, shown in Fig. 15.4, have been developed from parallel bridging models by introducing a hierarchy concept. One path aims at enhancing the performance accuracy of parallel bridging models. This is done by exploiting the concepts of processor locality and block-oriented communications. In doing this, solution are exploited which mimic those of hierarchical memory models. A different path, discussed in Sect. 15.4.3, explores the relationships among PBMs and HMMs, by developing equivaleces and emulation procedures that turn bulk-parallel algorithms into external memory ones. As we will see in Sect. 15.4.4, there are some results which show a possible convergence of the two approaches.

Because of the initial aim of modeling parallel computation, the main line of development of PBMs is directed towards a more accurate model of parallel locality effects. Most parallel architectures exploit regular, hierarchical structures which reward local data access. Even without making explicit geometric or implementation assumptions about the network, parallel bridging models can be refined by including parameters that indirectly reflect actual communication behavior.

The BSP communication model is simple and abstract enough to be refined this way. We describe two different extensions of BSP, which account

for (i) effects due to message length and (ii) for the relationship between network size and parallel overhead in communications.

The BSP* Model. In real interconnection networks, communication time is not independent of message length. The combination of bandwidth constraints, startup costs and latency effects is often modeled as a linear affine function of message length. BSP disregards this aspect of communication. The number of exchanged messages roughly measures the congestion effects on the network.

Counting non-local accesses is a first-order approximation that has been successfully used in external memory models like the PDM. On the other hand, PDM uses a block size parameter to measure the number of page I/O operation. To model the practical constraint of efficiency for real communications, the BSP* model [75] has been introduced in 1996.

BSP* adds a *critical block size* parameter b, which is the minimum size of data for a communication to fully exploit the available bandwidth. The cost function for communications is modified to account both for the number h_t of message start-ups in a phase, and for the communication volume s_t (the sum of the sizes of all messages). Each message is charged a constant overhead, and a time proportional to its length in blocks. Superstep cost is defined as $w_t + g(s_t/b + h_t) + L$, often written as $w_t + g^* \cdot (s_t + h_t \cdot b) + L$, where $g^* = g/b$. The effect on performance evaluation is that algorithm that pack information when communicating are still rewarded like in BSP, but high communication volumes and long messages are not. Thus the BSP* model explicitly promotes both block-organized communications and a reduced amount of data transfers, the same way external memory models do for I/O operations.

D-BSP Model. The BSP model inherits from the PRAM the assumption that model behavior is independent of its size. While we can easily account for a general behavior by changing parameter values (e.g. adding processors to a bus interconnection leads to larger values of g and L), there is no way we can model more complex situations where the network properties change according to the part of it that we are using. This is an intentional trade-off of the BSP model, but it can lead to inaccurate cost estimates in some cases. We mention two examples.

- Networks with a regular geometry, like meshes or hypercubes, can behave quite differently if most of the communication traffic is local, as compared to the general case.
- Modern cluster of multiprocessors and multiple-level interconnections cannot be properly modeled with any value of g, L, as shared memory and physical message-passing communications among different kinds of connections imply very different bandwidths and overheads.

De La Torre and Kruskal [240] introduce the *decomposable* BSP (D-BSP), which rewards locality of computation by allowing hierarchical decomposition

of the machine into smaller BSP-submachines. The g and L parameters of BSP are replaced with two functions, $\mathcal{G}_m$ and $\mathcal{L}_m$, of the submachine size m.

During algorithm execution, the computation can be recursively split, and smaller subproblems can be assigned to different submachines. The subcomputations are still D-BSP computations, which proceed independently until they merge together again. Their computational cost is the maximum of the costs, each one being evaluated with appropriate g, L values. The actual shape of $\mathcal{G}_m, \mathcal{L}_m$ controls the advantage of decomposing the computation into local subcomputations.

This abstract way of accounting for parallel locality avoids directly dealing with the geometry of the interconnection structure, which appears only by means of its characteristic functions. D-BSP thus adds the power to explicitly evaluate architectural effects on communications due to the geometry of the network, or due to its implementation.

A first comparison among the D-BSP and BSP models can be found in [111]. Meyer auf der Heide and Wanka [75] investigated the relationships among the BSP* and the D-BSP models. Bilardi and others [126] also examine the D-BSP model, concluding that it offers the same design advantages of BSP, but has higher effectiveness and portability over realistic parallel architectures. They show results for the family of functions $\mathcal{G}_m, \mathcal{L}_m$ of the form $C \cdot (n/2^i)^\alpha$, where $m = 2^i, \quad (0 \leq i \leq \log n)$ and $(0 < \alpha < 1)$. These functions capture a wide family of commonly used interconnection networks with n nodes, including multidimensional arrays.

15.4.3 Emulation in External Memory

A classical result of emulation of parallel algorithms using external memory techniques is the PRAM emulation algorithm in [192]. The result is described in Chapter 3. It is an asymptotically good emulation, but the asymptotic complexity hides quite large constant factors, as an external-memory sorting of the whole memory is required at each PRAM execution step.

It is clearly a fundamental issue to distinguish those emulations that are only asymptotically good, from those that can be practically exploited.

The emulation of PBM algorithms in external memory models has been shown to improve over known external memory algorithms in some cases, at least from the point of view of the abstract I/O complexity. The topic of external memory algorithm design is addressed in other parts of the book. Here we want to underline the connections among the two fields and the option to merge the two approaches into a more general, hierarchy-aware one.

Sequential BSP-like emulation. A simple simulation algorithm is presented in [692] by Sibeyn and Kaufmann. The emulation is performed by a sequential external-memory machine, simulating a number v of virtual BSP processors.

Once for each superstep, the computational context of each virtual processor is loaded into main memory in turn. A computational context consists of the memory image and message buffers of a virtual processor. Local computation and message exchange are emulated before switching to next processor.

Efficient simulation needs picking the right number v of virtual BSP processors with respect to the emulating machine. By implementing communication buffers using external memory data structures, we can derive efficient external memory algorithms from a subclass of BSP algorithms, those that require limited memory and communication bandwidth per processor. There are interesting points to note.

- the use of a *bandwidth gap* G parameter, measuring the ratio between the instruction execution speed and the I/O bandwidth,
- the introduction of a notion of x-optimality, close to that of c-optimality, which relates the number of I/O operations of a sequential algorithm with those of an emulated parallel algorithm for the same problem,
- the fact that the BSP* messages have a cost depending on their length in blocks helps in determining a relationship among the parallel algorithm and its external memory emulation.

Parallel emulation. In [242] the emulation results hold under more general assumptions. To evaluate the cost of *parallel emulation*, the EM-BSP* *model* is defined. EM-BSP* is a BSP* model extended with a secondary memory which is local to the processing nodes, see Fig. 15.5. Alternatively, we can see it as a PDM model augmented with a BSP* interconnection and a superstep cost function. In addition to the L, g, b, p parameters of BSP*, we find also local memory size M, the number (per processor) of local disks D, the I/O block transfer size B (which is borrowed from the PDM model) and the computational to I/O capacity ratio G as in the simpler simulation.

The emulation of BSP* algorithms proceeds by supersteps, but each emulating processor loads from disk a set of the virtual processors (with their needed context data) at the same time, instead of a single one. The emulation procedure can run sequentially in external memory, or in parallel, where the emulating machine is modeled using EM-BSP*. A reorganization algorithm is provided to perform BSP message routing in the external memories using an optimal amount of I/O.

Like in [692], the result in [242] exploits the BSP* cost function to simplify the emulation algorithm. BSP*, BSP and CGM algorithms (by reduction to BSP*) can be emulated if they satisfy given bounds on message sizes and of the memory used by the processors.

The c-optimality criterion is refined, taking into account I/O, computation and communication time of the emulated algorithm. We thus have a metric to compare EM-BSP* algorithms with the best sequential algorithms known.

Model		additional parameters and features
BSP*	b	critical block size
	$g*$	reduced message cost (g/b)
D-BSP	$\mathcal{G}_m$	g as function of submachine size m
	$\mathcal{L}_m$	L as function of submachine size m
CGM		*constraint on communication steps*
EM (PDM)	M	(node local) memory size
	D	(per node) number of disks
	B	block size for disk I/O
all models	G	*ratio of I/O and computation*

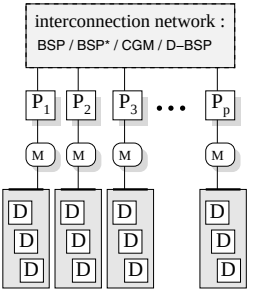

Fig. 15.5. The common structure of the combined parallel and external-memory models, and a summary of parameters used in the models, beyond those from BSP.

Parallel and serial external-memory emulation of bulk parallel algorithms is still under development [243, 244]. New results focus more on the CGM model, which is seen as a submodel of BSP with a known kind of routing relation (see Sect. 15.4.1), which allows a more efficient emulation. Despite technical improvements, the overall structure of the model and of the simulation is essentially unchanged from those of Fig. 15.5. These new works also identify an interesting subspace of algorithmic parameters where parallel external memory execution is efficient. Simulation has thus been used to obtain new or improved complexity results for several external memory problems.

15.4.4 Algorithms Designed for Parallel Hierarchical Models

In this section, we briefly survey two works which exploit mixed models of computation to develop parallel-external memory algorithms. The first work pre-dates most of the results we have previously presented. Aggarwal and Plaxton [16] define a multi-level storage model, made up of a chain of hypercubes of increasing dimension $0 \leq d \leq a$. A set of four primitive operations is defined on such a structure, which includes a scan operation, two routings and a shift of sub-cube data. A hypercube of dimension $b > a$ is then made up of the smaller ones, using bounded-degree networks to connect them through a subset of their nodes. The networks are supposed to compute prefix and scan operations in $O(b) = O(\log p)$ time. Due to its complexity, the model was never further developed despite a promising result on sorting.

Apart from the details of the model and of the sorting algorithm, in [16] it is indeed interesting to note how a parallel, hierarchical data space is defined on which to compute. The authors choose a set of primitive operation that can be practically implemented both in external memory and in parallel. This choice allows a certain degree of flexibility in choosing which levels of the computation to map to external memory, and which ones to perform in parallel.

Newer approaches, following the path of Fig. 15.4 (page 333), are based on external-memory extensions of BSP-like models. Dehne and others [246]

use a D-BSP interconnection structure in the configuration of Fig. 15.5, thus taking into account the hierarchy of the communication network and two levels of physical memory. This kind of models is of great practical interest for cluster and grid computing. The set of parameters in the composite model is essentialy the same of the sequential and parallel emulation approaches (table in Fig. 15.5). In particular, we now have a pair of g, G parameters relating computation time respectively with communication and I/O costs.

In [246], a set of basic primitives for sorting, merging and broadcasting is developed on the EM parallel model, by carefully composing D-BSP and PDM algorithms. The other result shown is a geometric algorithm following the distribution sweeping approach (see Chapter 6). It performs input partitioning to execute in parallel local external-memory computations. Intermediate results of local computations need to be exchanged, and composing the local and parallel parts of the algorithm is not simple, as the sweep paradigm does not allow a simple, one-shot input decomposition.

The solution devised is recursive in nature, and it exploits an orthogonal partitioning of the input and of the intermediate results of the recursion, such that the algorithm takes maximum advantage of parallel locality, and all the generated subproblems are independent and can be assigned to smaller subclusters.

15.5 Software Tools

In this section we will survey a set of software tools and programming models that can be used to exploit parallel hierarchical architectures. We classify them according to two main principles, the kind of hierarchy they exploit (either the parallel hierarchy, the memory hierarchy or both), and the number of levels they actually manage, which is two or three. Corresponding to the need for a unifying hierarchical model, we will see that there is a lack of software tools which span across multiple levels and different hierarchies. As a consequence, writing programs that fully exploit hierarchical parallel architectures is a difficult and error-prone task, where a large part of the effort is spent in tuning and debugging activities. In Sect. 15.5.1 we survey tools targeted at one of the two hierarchies, and spanning across two levels. Sect. 15.5.2 presents software systems that exploit parallelism and secondary memory, but only deal with a 2-level parallel hierarchy. Sect. 15.5.3 reports about recent proposals and experiments about extending existing parallel programming models, and developing new ones, which can cope with architecture hierarchy at least within SMP clusters.

15.5.1 Tools for Managing Parallelism or External Memory

Software tools of this first kind manage only two levels of a hierarchy. Since the mangement of parallelism and I/O are in principle completely separate,

these tools can be combined within the same environment to exploit architectures which correspond to the EM parallel models of Sect. 15.4.3 and 15.4.4. The work in the two separate fields of external memory and parallel programming is mature enough to have already produced some widely recognized standards.

Parallel Programming Libraries. There are two main parallel programming paradigms, which fit the two extremes of the MIMD architectural class, the distributed memory paradigm and the shared memory paradigm.

In the *message passing paradigm* each process has its local data, and it communicates with other processes by exchanging messages. This *shared nothing* approach corresponds to the abstraction of a DM-MIMD architecture, if we map each process to a distinct processor.

In the *shared-memory programming paradigm*, all the data is accessible to all processes, hence this *shared-everything* approach fits perfectly the SM-MIMD class of architectures. The programmer however has to formulate race-conditions to avoid deadlocks or inconsistencies.

For both paradigms, there is one official or de facto standard library, respectively the *message-passing interface* (MPI) standard, and the OpenMP programming model for shared memory programming. In both, MPI and OpenMP, possible hierarchies in the parallel target machine are not considered. They assume independent processors, either connected by an interconnection network or by a shared memory. Of course, there are approaches to incorporate hierarchy sensitive methods in both libraries. We will present some of them in Section 15.5.3.

Message-Passing-Interface MPI. In 1994, the MPI-Forum unified the most important concepts of message-passing-based programming interfaces into the MPI standard [547]. The current, upward compatible version of the standard is known as MPI-2 [548], and it specifies primitive bindings for languages of the C and Fortran families.

In its simplest form, an MPI program starts one process per processor on a given number of processors. Each process executes the same program code, but it operates on its local data, and it receives a *rank* (a unique identifier) during the execution, that becomes its address w.r.t. communications. Subject to the rank, a process can execute different parts of the program. This single program multiple data (*SPMD*) model of execution actually allows a generic MIMD programming model.

There are MPI implementations for nearly all platforms, which is the prerequisite for program portability. Key features of the MPI standard include the following, and those described on page 342 about the I/O.

Point-to-point communication: The basic MPI communication mechanism is to *exchange messages between pair of endpoint processes*, regardless of the actual network structure that delivers the data. One process initiates a send operation and the other process has to start a receive operation in order to start the data transfer.

Several variants of the basic primitives are defined in the standard, which differ in the communication protocol and the synchronous/asynchronous behavior. For instance, we can choose to block or not until communication set-up or completion, or to use a specific amount of communication buffers.

These different options are needed both to allow optimized implementation of the library and to allow the application programmer to overlap communication and computation.

Collective operations: *Collective communications* involve a group of processes, each one having to call the communication routine with matching arguments, in order for the operation to execute.

Well-known examples of collective operations are the *barrier synchronization* (processes wait for each other at a synchronization point), the *broadcast* (spreading a message to a group of processes) or the scan operation.

One-sided Communications: With *one-sided communication* all communication parameters for both, the sender and the receiver side, are specified by one process, thus avoiding explicit intervention of the partner in the communication. This kind of remote memory access separates communication and synchronization. Remote write, read and update operations are provided this way, together with additional synchronization primitives.

OpenMP. The *OpenMP-API* [593] is a standard for parallel shared memory programming based on compiler directives. Directives are a way to parameterize a specific compiler behavior. They preserve program semantics, and have to be ignored when unknown to a compiler. Thus they are coded as `#pragma` statements in C and C++, and are put within comments in Fortran. OpenMP directives allow to mark parallel regions in a sequential program. This approach facilitates an incremental parallelization of sequential programs.

The sequential part of the code is executed by one thread (master thread) that forks new threads as soon as a parallel region starts and joins them at the end of the parallel region (*fork-join model*). OpenMP has three types of directives.

Parallelism directives mark parallel regions in the program.

Work sharing directives within a parallel region divide the computation among the threads. An example is the *for/DO* directive (each thread executes a part of the iterations of the loop).

Data environment directives control the sharing of program variables that are defined outside a parallel region (e.g. shared, private and reduction).

Synchronization directives (barrier, critical, flush) are responsible for synchronized execution of several threads. Synchronization is necessary to avoid deadlocks and data inconsistencies.

External Memory Programming Libraries. There are libraries designed to simplify processing of external-memory data structures. While the main reference model, PDM, is parallel, these libraries usually only support sequential algorithm operation over parallel disks. To use these libraries in a parallel setting, we can either use independent physical disks, or use independent data structures on a shared device (with a possible performance loss).

TPIE. TPIE [57] is a library developed as a programming tool to simplify the implementation of algorithms based on the PDM model. The assumption of the TPIE approach is that all operations and access methods exploit the best known EM algorithm for the problem, and that any TPIE-based program can immediately benefit of theoretical and practical improvements in EM algorithms, as soon as they are propagated to the library.

TPIE initially provided data structures and algorithms to solve batched problems, providing a strongly stream-oriented interface to the data. A recent work by Arge and others [66] presents an extension of TPIE to deal with random-access data structures.

Common operations on streams are provided (e.g. sort, merge, distribution, permutation) as well as on simple external-memory data types (matrices, stacks). A flexible, general support for external memory trees allows to code several different tree management strategies.

The fundamental components of TPIE are the memory manager, which controls in-core memory utilization, and the *block transfer engine* (BTE), the software kernel that moves blocks of data from physical devices to main memory and back. Two separately designed BTEs deal respectively with streamed and random accesses to the disk, interacting with a common memory manager. The BTEs have different and interchangeable implementations based on the UNIX *stdio* functions, on blocked read/write calls and on memory-mapped I/O.

The current implementation of the library is completely sequential, as each BTE manages a single physical disk, and BTEs on different processors do not cooperate. Interprocess coordination is left to the user like it is in PDM algorithms.

LEDA-sm. LEDA [570] is a commercial library of data structures and algorithms for combinatorial and geometric computation. LEDA-sm is a secondary memory extension [229] which is publicly available under the GPL software license. It provides external data structures (e.g. arrays, queues, trees, strings) with basic operations, and algorithms that work on these structures. Like TPIE, the implementation relies on a library kernel, the EM manager, which implements a PDM abstraction and programming interface over concrete disk devices.

15.5.2 Tools for Parallel-External Programming

In this section we present two tools that allow to implement external-memory aware, parallel programs.

VIC*. The VIC* compiler [214] is a compiler for the C* language, an extension of C with virtual-memory data parallel constructs. The VIC* compiler produces code which interfaces to an implementation of the PDM model.

It allows to fully exploit the PDM model, using parallel disks and processing elements. The user can control the number of computing units and data-server processes that are set up on the target machine. The run-time support of VIC* has been implemented over a set of different sequential and parallel architectures, exploiting existing conventional, networked and parallel file systems. Widely available libraries, including MPI, are used to support the communication.

VIC* has been used to evaluate the actual performance of sequential and parallel PDM algorithms. It shows the effectiveness of the PDM model, as the tested external-memory algorithms are mainly computation bound, whereas their main-memory counterparts are severely I/O bound at the same problem size [214].

While VIC* programs exploit available parallel resources, there is still no model for the communication part. An interesting result in this view is reported in [90] about the problem of external memory, parallel FFT. In [90], the problem is solved by using the dimensional decomposability of the FFT to devise a parallel partitioning of the out-of-core computation.

MPI-IO. The MPI-2 standard includes the specification of a parallel I/O programming interface. Programs written using MPI-IO can exploit message passing parallelism and a shared disk space, while remaining largely portable. A full discussion of parallel file systems is not appropriate here, so we summarize the MPI-IO approach and its rationale.

A typical parallel I/O scenario is that of multiple processors in a MIMD machine (Fig. 15.1b,c) trying to access different parts of a single large file. Shared memory architectures often use centralized I/O subsystems, and the target is to minimize contention due this bottleneck. Distributed memory architectures, on the other hand, usually have local disks in each processing node. In order to exploit these disks as a single storage support within a parallel program, data blocks are sent through the network from hosting nodes to the requesting ones. In both cases, shared and distributed memory, the solution to the performance problem of I/O lies in aggregating several requests to serve them efficiently.

The UNIX-like semantics of most file systems does not allow this transformation [722]. Indeed, the gain is significant if the program explicitly gives information about collective I/O (parallel, logically synchronized I/O requests from a set of processors). MPI-IO provides this interface. MPI *derived datatypes* are a portable mechanism to specify the memory layout of a data

structure. They allow MPI functions to minimize communication overheads, and to automatically compact non-contiguous data structures. MPI-2 has extended the use of MPI datatypes from communication to parallel I/O. Since MPI-IO also offers collective and asynchronous I/O functions, there is plenty of room for optimizations.

ROMIO [722, 723] is a public domain implementation of the standard. It is based on a virtual device interface, ADIO, which connects to most sequential, networked and parallel file systems in current use. [722, 723] show that the following two optimizations are fundamental in boosting parallel I/O performance.

Data sieving: Separate asynchronous requests are reordered and merged into bigger ones. In doing this we can afford to read a certain amount of extra data in the "holes", in order to reduce the number of separate I/Os.

Collective request merging: Parallel I/O requests to the same file often address small different regions of it, according to complex patterns which depend on the application (e.g. processor i reads blocks $i+k\cdot j, j = 0, 1, \ldots$). Merging together all of these patterns we can identify a much simpler I/O pattern at the hosting nodes, which is used to satisfy all the requests by means of a data reorganization phase.

We note two features of the library which are also of more general relevance. The first one is that the same concepts are used for communication and I/O programming interfaces: data types, contexts, asynchronous versus synchronous operations. The second one it that most of the optimizations that are possible with MPI-IO rely on exploiting a form of global architecture-level caching. Data from the external memory level are loaded into memory buffers which are shared by hardware and software means.

15.5.3 Tools for Parallel-Hierarchical Programming

In this section we survey some proposed approaches to the problem of producing efficient programs on hierarchical parallel architectures like SMP clusters. Starting from the shared memory and distributed memory programming standard, we can choose one and try to develop optimizations and extensions, we can try to merge the two, or we can develop new, different programming models. Several approaches are still proposals, but, looking at the available experimental results, those solutions that try to hide the architecture hierarchy to the programmer do not produce the expected performance gain.

Hierarchical Optimizations for MPI. The message-passing paradigm does not consider the hierarchical architecture of SMP clusters. In the following, we present two approaches for adapting MPI to SMP clusters that try do avoid this inefficiency.

Shared-Memory Communication. This approach improves the communication between processors that reside in the same node by using the shared-memory for point-to-point communication. When a message is sent, the system detects if the target process works on a processor that resides in the same node. If this is the case, the message will be delivered through shared memory, instead that over the network. Reducing the number of message copy operation is a well known issue to reduce the host overhead and latency of message-passing communication. For inner-node communication, in-memory copying is the largest part of the message delivery cost, so such an optimization is even more important.

In [712] optimizations are presented of inter-node and inner-node communication for a special MPI implementation, that works on PC-based SMP clusters. Performing an inner-node communication initially requires two message copies, to go from the memory space of one process to that of another one by means of the UNIX kernel primitives. Single-copy operation is achieved by building a dedicated kernel primitive, that writes directly into the receiver's memory.

In order to test the library, the authors performed experiments using the *NAS Parallel Benchmark 2.3* [86]. This benchmark suite is a set of 8 programs designed to help evaluate the performance of parallel supercomputers.

In [712], NAS results on an SMP cluster are compared with those on a cluster of uni-processors with the same number of processors. The SMP cluster achieved 70-100% of the performance of the uni-processor cluster. Intuitively, the SMP clusters should perform better, thanks to the inner-node communication. Several differences between the two clusters can all together explain the results. Communication latency is higher for a process on the SMP, as there is a single, shared network interface per node. Cluster-level synchronization mechanisms are realized by means of messages, and their cost is dominated by inter-node communication delay. Thus, for all applications which do a lot of synchronizations, like those of the NAS benchmark, the inner-node communication performance is wasted.

In conclusion, the approach can improve the average point-to-point communication time, but it it is not guaranteed to improve the overall performance of a program, when compared to that of a cluster of single-processor machines. Indeed, the problem is that the programmer is not forced to consider the SMP cluster architecture at all during the design of an application. The SMP cluster is seen as *flat* parallel machine, thus there is no way to match the program structure with the real architecture.

Threads Only MPI. The *threads only MPI* (*TOMPI*) [248] is an MPI for uniprocessor and SMP workstations. The aim is to make the development of MPI programs on workstations less time-consuming. Most standard MPI implementations, for the sake of portability, use UNIX processes and UNIX domain sockets. When working on a single workstation, this method is resource-inefficient and involves an unneeded overhead. TOMPI uses a source code

translator to rewrite MPI program into thread-based programs, which can be executed on an SMP workstation.

This approach seems to have the potential to be more efficient than the one based only on shared memory communication. It improves the speed of communications and avoids the large memory overhead due to processes. On the one hand, with TOMPI it is possible to execute MPI programs with hundreds of MPI processes on a single workstation, without bringing the system down. On the other hand, there is no extension of the approach to SMP clusters yet, even if converting processes to threads is an interesting opportunity for this kind of architectures.

Distributed Shared-Memory Programming with OpenMP. In the following, we present two approaches that try to adapt OpenMP to SMP clusters. The main issue for this approach is that there is either a need for a global shared memory in physical distributed environment or OpenMP has to be extended with data distribution facilities.

Software Distributed Shared Memory. Software distributed shared memory (*SDSM*) systems are libraries that provide a global address space for physically distributed memory machines. We can translate OpenMP directives into appropriate calls to the SDSM system. An example of this approach is described in [413]. The result of the source-to-source translation is a standard C/C++ or Fortran program which can be compiled and linked with the SDSM system TreadMarks [41].

Even in this case, the communication system is modified to exploit the hardware shared memory within the SMP nodes. Experimental testing with several algorithms showed that the performance of the modified software distributed shared memory was much better than the original ThreadMarks library. Nonetheless, performance was still worse than that achieved by an MPI implementation of the programs. The speedups obtained were only 7-30% of those achieved by the MPI versions. The reasons are the overhead from coherence-maintenance network traffic, and the fact that the SDSM system does not exploit application-specific data access patterns, because only at run-time it is known whether communication will happen through the network or not. Again, the issue is that the method does not allow the programmer to explicitly address the hierarchical structure of the machine at design and at compile time.

A more promising approach is the *compiler directed SDSM* one [662], which is a two-step optimization.

In a first step, the OpenMP compiler inserts memory coherence code primitives, called *check code*, to keep the node-distributed memory consistent. There are three types of check codes. Two of them ensure that the data is valid before a read or write of shared data, the third is responsible to inform the other nodes that data has been changed by a shared write.

In the second step, the compiler analyzes parallel regions in order to optimize communication and synchronization by removing unnecessary check codes. The following optimization strategies are applied.

Parallel extent detection. Memory coherence code only has to be used in parallel regions. Therefore, the compiler can remove the check codes outside parallel regions and in the static extent of parallel regions.

Redundant check code elimination. Flush directives are responsible for giving all threads a consistent view of the memory. They are executed implicitly at barrier synchronizations, at the end of work sharing constructs and at references to volatile variables. Therefore, check codes after a write may be delayed until the thread reaches a flush directive and check codes before a read or write may be redundant if the data is already available by the preceding read check at the same location. The compiler performs a data-flow analysis of the statements in the parallel regions to determine the earliest possible read check code, and the latest possible write check code. All others are redundant and can be removed.

Merging multiple check codes. Arrays are very often accessed contiguously within a loop structure. The corresponding check codes may be moved outside the loop and simultaneously converted into a single one. This reduces the number of check code calls. In the following example `a` and `b` are shared arrays.

```
for (i=0; i<n;i++) a[i]=c*b[i];
```

The compiler inserts the check codes into the loop as follows.

```
for (i=0; i<n;i++) {
    check_before_read(&b[i], size);
    check_before_write(&a[i], size);
    a[i]=c*b[i];
    check_after_write(&a[i], size);
}
```

Since the loop does not contain any flush directive, the check codes can be moved outside the loop.

```
check_before_read(&b[0], n*size);
check_before_write(&a[0], n*size);
for (i=0; i<n;i++)  a[i]=c*b[i];
check_after_write(&a[0], n*size);
```

Data-parallel communication optimization. It is also possible to improve the program by using data-parallel compilation techniques. For example, the compiler should determine data mappings of arrays that accessed within a loop in such a way that iterations of the loop can be done locally, on the nodes

where the data is stored. Since the number of threads is not known at compile time, calls to a data mapping runtime library are inserted to compute loop bounds and data that must be communicated. In this setting, the data is stored locally and check codes can be removed.

Collective communication optimization. Inter-node communication is necessary to implement a reduction operation on variables defined in the data scope attribute of a parallel region. It can be performed efficiently using a collective communication library. The execution starts after the local reduction at the end of parallel regions or after work-sharing directives.

Distributed OpenMP. A different approach to adapt OpenMP to SMP clusters is suggested in [546]. The authors propose the *distributed OpenMP*. This extension of OpenMP with data locality features provides a set of new directives, library routines and environment variables. One data-distribution extensions is the `distribute` directive with which it is possible to partition an array over the node memories. For performance reasons, the threads should work on local array elements. Hence, the user must distribute the data in order to minimize remote data accesses. Another proposed extension is the `on home` directive in a parallel region. With this directive, it is possible to perform a parallel loop over a distributed array without redistributing the array. The threads of a node perform the iterations for the array elements that reside in their local memory. Further extensions are library routines and environment variables that provide specific numbers of the run-time instance of the SMP cluster, like for example the number of involved nodes or processors per node. Disadvantages are that programs get more complex, and the user has to take care about an efficient data decomposition. Since we are providing more information to the compiler, after adding the new directives to an OpenMP program a redesign step, and a performance tuning phase have to be performed.

Hybrid Programming with MPI and OpenMP. The idea of the *hybrid programming model* is to use message passing between the SMP nodes, and shared memory programming inside the SMP nodes. The structure of this model fits exactly to the architecture, therefore, the model has potential to produce programs with significant performance improvement. But it is also obvious that the model is more complicated to use, and that there may arise unpredicted performance problems, because of the simultaneous usage of the two programming models. There are several possibilities for choosing libraries for each model, but it is straightforward to combine the de facto standards *MPI and OpenMP*. In the following we give an overview of the different approaches to the production of hybrid programs, with no emphasis on technical details. We also survey some performance evaluations that compare hybrid programs with pure MPI ones.

The general execution scheme uses one process in each node, to handles communications by means of MPI primitives. Inside the process, multiple threads compute in parallel. The number of threads in a node is equal to the

number of processors in that node. The base for the design of an efficient hybrid program is an efficient MPI program. According to [171], there are two approaches to incorporate OpenMP directives into MPI programs, the fine-grain and the coarse-grain approach.

Fine-Grain Parallelization. The hybrid *fine-grain parallelization* is done incrementally. The computational part of an MPI program is examined, and the loop nests are parallelized with OpenMP directives. Therefore, the approach is also called loop-level parallelization. Clearly, the loops must be profiled, and only loop nests with a significant contribution to the global execution time are selected for OpenMP parallelization.

Some loop-nests can not be parallelized directly. If they are non negligible, the developer can try to transform them into parallelizable loops. Techniques like loop exchange and loop permutations, and introduction of temporary variables, can often avoid false sharings and reduce the number of synchronizations.

Performance of Fine-Grain Hybrid Programs. In [171, 172, 173, 200] investigations to measure performance of fine-grain hybrid programs are presented. A comparison is shown of the performance achieved by a hybrid and a pure MPI version of the NAS benchmark [86] on a SMP cluster. An important subject of [171, 172, 173, 200] is the interpretation of the performance measurements, in order to understand the behavior of the hybrid programs and their performance. Experiments were made on a PC-based SMP cluster with two processors per node and on IBM SP cluster systems with four processors per node.

The comparison among the two kinds of models for SMP clusters shows no general advantage of one over the other. Depending on the characteristics of the application, some benchmarks perform better with the hybrid version, others perform better with the pure MPI version. The following aspects have influence on the performance of the models.

Level of shared memory parallelization. The more of the total computation can be parallelized, the more interesting is the hybrid approach. The size of the parallelized sections (OpenMP) compared to the whole computation section must be significant.

Communication time. It depends on the communication pattern of an application, and on the differences between the two models concerning latency, bandwidth, and synchronization time. If more processes share one network interface, then the latency for network accesses increases, but the per process bandwidth increases too. If there is only one process per node, the latency is low, but the process cannot transfer data fast enough to the network interface to fully exploit the maximum network bandwidth. Therefore, the pure MPI approach performs better if the application is bandwidth limited, and it is worse for latency limited applications.

Memory access patterns. The memory access patterns are different for the two models. Whereas MPI allows to express multi-dimensional blocking, OpenMP does not. To achieve the same memory access patterns, rewriting of loop nests is necessary, which may be very complex.

Performance balance of the main components. (processors, memory and network) can offset the communication/computation tradeoff. If the processors are so fast that communication becomes the bottleneck, then the actual communication pattern decides which model is best. If, on the other hand, computation is the bottleneck, then MPI seems to be always the best.

Coarse-Grain Parallelization. In this approach a single program multiple data style is used to incorporate OpenMP threads into MPI programs. OpenMP is used to spawn threads immediately after the initialization of the MPI processes in the main program. Each thread itself is acting similar to an MPI process. For threads there are several issues to consider:

- The *data distribution between the threads* is different from that of MPI processes. Because of the shared memory, it is only necessary to calculate the bounds of the arrays for each thread. There has to be a mapping from array regions to threads.
- The *work distribution between the threads* is made according to the data distribution. Instead of an automatic distribution of the iterations, some calculations of the loop boundaries depending of the thread number define the schedule.
- The *coordination of the threads* means managing critical sections by either the usage of OpenMP directives, like `MASTER` or thread library calls like `omp_get_thread_num()`, to construct conditional statements.
- *Communication* is still done by only one thread.

As far as we know, the coarse-grain approach has been proposed, but there are no results yet. We can compare it with TOMPI, as both methods convert MPI processes to threads. However, TOMPI programs on SMP clusters do not share data structures common to all the processes, as they would do in a coarse-grain parallelization.

High-Level Programming Models. Besides the programming libraries and paradigms above, there are some programming models for SMP clusters that try to build a higher level of abstraction for the programmer. All these models are based on the hybrid programming paradigm where threads are used for the internal computation and message passing libraries are used to perform communication between the nodes.

SIMPLE Model. The significant difference between *SIMPLE* [82] and the manual hybrid programming approach above lies in the provided primitives for communication and computation.

The computation primitives comprise data parallel loops, control primitives to address threads or nodes directly, and memory management primitives.

Data parallel loops: There are several parallel loop directives for executing loops concurrently on one or more nodes of the SMP cluster, assuming no data dependencies. The loop is partitioned implicitly to the threads without need for explicit synchronization or communication between processors. Both block and a cyclic partitioning is provided.

Control: With this class of primitives, it is possible to control which threads are involved in the computation context. The execution of a block of code can be restricted to one thread per node, all threads in one node, or to only one thread in the SMP cluster.

Memory management: A heap for dynamic memory allocation is managed in each processing node, and can be used by the threads of that node via the `node_malloc` and `node_free` primitives.

SIMPLE provides three libraries for communication. There is an inter-node-communication library, an SMP node library for thread synchronization, and a SIMPLE communication library build on top of both. The SMP node library implements the three primitives *reduce*, *barrier* and *broadcast*. It is based on POSIX threads. Together with the functionality of the inter-node-communication library, it is possible to implement the primitives *barrier, reduce, broadcast, allreduce, alltoall, alltoallv, gather, and scatter* that are assumed to be sufficient for the design of SIMPLE algorithms. The use of these top-level primitives means using message passing between nodes and shared memory communication within the nodes.

Hybrid-Parallel Programming with High Performance Fortran. High Performance Fortran (*HPF*) is a set of extensions to Fortran that enables users to develop data-parallel programs for architectures where the distribution of data impacts performance. Main features of HPF are directives for data distribution within distributed memory machines and primitives for data parallel and concurrent execution. HPF can be employed on both distributed memory and shared memory machines, and it is possible to compile HPF programs on SMP clusters. However, HPF does not provide primitives or directives to exploit the parallel hierarchy of SMP clusters. Most HPF compilers just ignore the shared memory within the nodes and treat the target system as a distributed memory machine.

One exception is presented in [106]. Therein, HPF is extended with the concept of *processor mappings* and the concept of *hierarchical data mappings*. With these two concepts, it is possible for the programmer to consider the hierarchical structure of SMP clusters. A product of this approach is the Vienna Fortran Compiler [105]. It creates *fine-grain hybrid programs* using MPI and OpenMP, starting from programs in an enriched HPF syntax.

Processor mappings: Beside the already existing *abstract processor array* that is used as the target of data distribution directives, *abstract node arrays* are defined. Together with an extended version of the `distribute` directive it is possible to construct the structure of an SMP cluster.

Hierarchical data mapping: In addition to the processor mappings it is necessary to assign data arrays to nodes and processors. The `distribute` directive is extended in the way that node arrays may appear as distribution targets. This defines an explicit inter-node mapping of the data. In contrast, the `share` directive is introduced in order to define an explicit intra-node mapping. The intra-node mapping controls the work sharing between the processors within a node.

Intrinsic functions: Two new functions are provided, that return the number of nodes and the number of processors in the SMP cluster. They are provided in order to support abstract node arrays whose sizes are determined at program startup.

The following is a sample code fragment for the use of the new directives and mappings. It defines a SMP cluster with four processors per node and distributes an array A equally over the nodes and processors.

```
!hpf$ processors P(2,8)                !abstract processor array
      real, dimension (32,16) :: A     !array of real
!hpfC nodes N(4)                       !abstract node array
!hpfC distribute P(*, block) onto N    !processor mapping
!hpfC distribute A(*, block) onto N    !inter-node mapping
!hpfC share A (block,*)                !intra-node mapping
...
```

`block` is a standard HPF distribution format and divides the concerned dimension into equal parts with respect to the distribution target. The asterisk defines that the whole dimension of the array will be mapped to the target elements.

KeLP2 Model. The *Kernel Lattice Parallelism 2 Model (KeLP2)* [81] is a C++ framework for implementing (irregular) block-structured numerical applications on SMP clusters. KeLP2 provides mechanisms to coordinate data decomposition, data motion and parallel control flow similar to HPF. Like HPF, it hides from the programmer low-level details like message-passing, processes, threads, synchronization and memory allocation. In contrast to HPF, KeLP2 performs no analysis of the code to make high-level restructuring, and it provides no automated data decomposition.

The underlying assumption is that the programmer knows best the structure of his (irregular) data and algorithm. KeLP2 provides him a framework and a methodology to define the decomposition, facilitating the construction of partitioning libraries. With respect to HPF, KeLP2 allows to overlap computation and communication.

KeLP2 supports three levels of control, the collective level (SMP cluster), the node-level, and the processor-level. Parallelism in programs is expressed at the node and the processor levels, while communication takes place in the collective (cluster) and node levels. The collective and the node levels

have their own data layout and data motion plan. Three classes of KeLP2 programming abstractions help to manage this mechanisms.

1. The *Meta-Data* represents the abstract structure of some facet of the calculation. It describes the data decomposition and the communication patterns.
2. *Instantiators* execute the program according to the information contained in the meta-data.
3. The primitives for *parallel control flow* are iterators, which iterate over all nodes, or over all processors of a specified node.

When comparing KeLP2 with SIMPLE, the latter provides lower-level primitives, does not support data-decomposition, and does not overlap communication and computation. KeLP2 is of narrower scope concerning the application domain, but it nevertheless enables a parallel specification that is less dependent on the implementation.

15.6 Conclusions

It is clear from Sect. 15.2 that we need theoretical models and new software tools to fully exploit hierarchical architectures like large clusters of SMP and future Computational Grid super-clusters.

The interaction among parallel bridging models and external memory models has produced several results, which we surveyed in Sect. 15.3, 15.4. The exploitation of locality effects in these two classes of models employs very similar solutions, that involve block-oriented cooperation and abstract modeling of the hierarchical structure. The intuition underlying theoretical and performance results on bulk parallel models, and their theoretical lesson, is that the simple exploitation of fine-grain parallelism at the algorithm level is not the right way to obtain portable parallel programs in practice.

However, there is still a lot of work to do in order to meet the need of appropriate computational models. Hierarchical-parallel models like those of Fig. 15.5 are already close to the structure of modern SMP clusters, and they are relatively simple to understand, yet algorithms can be quite hard to analyze. Composed models usually employ the full set of parameters of their parallel part, those of their memory part, and at least another one to asses the relative cost of I/O and communication operations. For the BSP derivatives we have described, this leads to seven or eight parameters.

Excessive complexity of the analysis and poor intuitive understanding are a limiting factor for the diffusion of computation models, as it was pointed out in [213]. There is no answer yet to the questions "can all these parameters be merged in some synthesis?" and "what are the four or five most important parameters?"

For these reasons the impact of sophisticated parallel computational models is still limited, while simple disk I/O models like PDM have been quickly

adopted to evaluate a corpus of external memory algorithms widely used in practice.

The situation is the same for what concerns software tools. Indeed, most of the efforts in parallel software development are spent in maintaining existing programming interfaces and optimizing them for new architectures. This approach is not effective or efficient for hierarchical parallel architectures. As we have reported, SMP cluster-enabled implementations of both MPI and OpenMP libraries are quite far from exploiting the potential performance of these machines. When programmers completely disregard the existence of a hierarchy, and we try to hide all optimizations in the library, it is often impossible to achieve program implementations with optimal performance.

A complementary approach is to compose existing libraries which manage a portion of the hierarchy. Most standard libraries only exploit two level of memory or parallel structure (e.g. MPI, OpenMP, TPIE). While it is possible to compose libraries that manage well-separated hierarchies (e.g. main memory/disks and main memory/shared memory), we have seen in Sect. 15.5.3 that the hybrid programming model, resulting from the empirical combination of MPI and OpenMP, leads to much harder problems.

Nevertheless, the hybrid model is the only parallel and hierarchical programming model accepted in practice, because it has the driving advantage of exploiting the new cluster architectures using existing, available tools. It has numerous drawbacks, though. Programs are more complex to design, implement, debug and maintain. Implementation and debugging are also complicated by the need for extensive performance analysis and tuning. The complexity and the amount of the interactions among the architecture, the algorithm and the software tools make it quite difficult to devise performance models for the resulting programs.

According to us, two main directions for future research are now still open. A first research and development track aims at simplifying the management of hybrid parallel programming, by extending existing flat approaches with ad-hoc optimizations, and with improvements to the compilation tools. Code translators, e.g. from MPI to hybrid structured code, and semi-automatic restructuring tools are two feasible solutions to ease and speed up hybrid software development.

Another option is to devise simpler parallel hierarchical models that are sound and intuitive, and thus can be used both for theory and as the base for a programming environment. Too cumbersome models are ruled out, as they fail in providing that kind of intuitive guidance that, even if not fully trustworthy, is essential for the acceptance of a programming model.

Looking at the literature, an important resource in this perspective is a set of common, basic operations that can be efficiently performed both in parallel and exploiting the memory hierarchy. More freedom in choosing the implementation level of the basic operations simplifies design and analysis of more complex algorithms, as well as the implementation of software tools.

Systems like SIMPLE and KeLP2 are close to this research path. Abstract, high-level operations simplify program writing, while still providing the tools with information about the best mapping to the architecture hierarchy.

Acknowledgments

We wish to thank all the participants to the GI-Dagstuhl-Forschungsseminar "Algorithms for Memory Hierarchies". In particular we would like to thank Florin Isaila and Kay Salzwedel for their help in the bibliographic research, and Peter Sanders, Jop Sibeyn, Ulrich Meyer, Rasmus Pagh, and Daniel Jimenez for several discussion which contributed to improve the quality of the paper.