Hierarchical Models and Software Tools for Parallel Programming

Massimo Coppola [a], Martin Schmollinger [b]

[a] Department of Computer Science, University of Pisa
coppola@di.unipi.it

[b] Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
schmolli@informatik.uni-tuebingen.de

Abstract This paper surveys current results about parallel programming on hierarchical architectures. We summarize the concepts and characteristics of modern architectures, which show a clear trend toward SMP cluster and grid-enabled solutions. These systems need computational and programming models. We present collected results about bulk-parallel bridging models (BSP, CGM and extensions), their emulation in external memory, and some proposed extension to parallel external-memory models. Some software tools already exist that allow to exploit part of the memory and communication hierarchies. We describe existing standards and their development track toward the exploitation of hierarchy-aware parallelism.

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 long-time recognized, only in the last years hierarchical parallel structures have gained importance, mainly as a result of the trend toward 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 already pointed out before (see for instance [1]). It is an open question if a single, unified model of both aspects exists, and if it is theoretically tractable. On the application edge, a single programming environment which includes support for both hierarchies is still lacking. Only if there are well-founded models and efficient tools we can really exploit a connection from algorithm design and complexity results to high-performance program implementation.

In this work 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 target is to investigate which software supports have been realized, and which ones are still missing, in order to exploit the full performance of modern high-performance cluster architectures. Even in this case, solutions emerge from merging results of different fields.

The paper is organized in three parts. The first one 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 concept of hierarchy associated with the exploitation of parallelism. We discuss the technological reasons and future expectations of current architectural trend.

The second part of the paper gives an overview on parallel computational models, exploring the connection among the so-called *parallel bridging models* (PBM) and external memory models, here mainly represented by PDM. There is a similarity of problems between bulk parallelism and block-oriented I/O, as both techniques try to efficiently exploit locality in mapping algorithmic patterns to a hierarchical structure. We discuss the issues of parallel computation models in section 3. In section 4 we survey definitions and results about PBMs, and works that emulate PBM algorithms in sequential and parallel external memory. This part ends presenting two works that exploit parallel hierarchical models for algorithm design.

The third part (section 5) gives an overview of software tools and programming models which may be used for program implementation. We consider libraries for parallel and external programming. Regarding parallel software tools, we focus on the approaches which support hierarchy-aware program development. We present tools and libraries relating them to the different levels of abstraction and structure found in part two. Section 6 summarizes the paper and draws conclusions.

2 Architectural Background

In recent years, a strong trend has emerged in the field of high performance computers towards two kinds of architectures, clusters of vector computers and clusters of scalar uni- and multiprocessors. Looking at the list of the fastest 500 supercomputers in the world, 90% of them belong to these two classes [2,3], with the latter steadily gaining more share. Translated into the taxonomy of Flynn this means that industry and research are focusing on only one class, the MIMD one (Multiple Instruction streams, Multiple Data streams). The MIMD class is subdivided in [4] according to the characteristic of the physical memory, into SM MIMD (shared memory MIMD) and DM MIMD (distributed memory MIMD).

The memory banks of a SM MIMD machine form a common address space. Clearly, the design issues are what protocols, and which structures have to be used to connect the processors with the memory banks of the system. Hardware access protocols are needed to avoid inconsistencies due to interference in the memory access between different processors, and the choice of consistency protocols is related to that of the network and to the expected performance. The more processors we put in such systems, the more complex the problem gets.

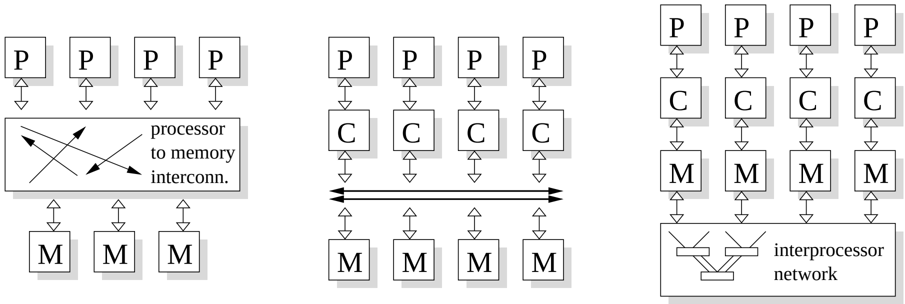

Figure1. General structure of DM and SM MIMD architectures. (a) Generic MIMD machines have multiple processors and memory banks (not necessarily the same number) – (b) example of SMP with multiple memory banks, processor caches and a bus interconnection – (c) example of DM MIMD architecture composed of multiple single-processor nodes and a fat-tree interconnection network.

The simplest and less expensive interconnection structure (w.r.t. circuital complexity) is a central bus, the most expensive one is a crossbar. A lot of systems use structures which are in between these two, as a compromise between practical scalability and performance. Multi-stage crossbars are getting more and more important with the increasing number of processors in SM 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.

Systems in which the access from a processor to some of the memory banks (or its local one) is faster than access to the rest of the memory are called NUMA systems (Non-Uniform Memory Access). By contrast, if the access times to the memory are the same for all processors, then we have a UMA machine (Uniform Memory Access). NUMA systems try to hide this hierarchy by applying appropriate cache strategies. SM MIMD architectures are often called SMP (Symmetric Multiprocessors), since the architecture is fully symmetric from the point of view of the running programs.

Contrary to the shared memory machines, there is no common address space in a distributed memory machine. Each processing node has its own address space and, therefore, it is up to the user to define efficient data-decompositions and data-exchange patterns for the applications. Likewise, DM machines have to confront with the issues concerning the interconnection structure among the processing nodes. Popular structures for DM machines are the hypercube, the 2D or 3D meshes, fat trees and multi-stage crossbars. DM architectures obviously belong to the NUMA architectural class, as each processing node has its local memory, which is immediately accessible, while it has to go through the (slower) network to access the memories of the other nodes. Hence, we have a memory hierarchy in DM MIMD machines too. On the other hand, DM MIMD architectures are less demanding with respect to the interconnection network, so they are much more scalable than the SM ones.

As we previously mentioned, clusters of symmetric multiprocessing nodes (SMP) are nowadays a more and more popular architecture for building supercomputers. At a different scale, these systems can be classified as DM and SM MIMD architectures, because SM MIMD processing nodes are connected together to form a larger DM 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, SMP clusters can be classified either as SM MIMD or as DM MIMD depending on the existence or not of a common address space for all the processors. However, even if the abstraction of a common address space is provided to the software, algorithms should try to exploit the local memory within SMP nodes as much as possible, in order to achieve a good performance. Thus, SMP clusters have 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, where supercomputers or clusters of workstations are connected with each other, result in even more levels and a less regular parallel hierarchy. Broadband connections, ranging from local area networks to geographic ones, add more levels to the hierarchy, with different communication bandwidth and latency [5, chapter 2].

Table1. Order of magnitude of parameters for different levels in a hierarchical parallel architecture

Layer	Bandwidth	Latency	Notes
CPU	10-100 GB/s	1-10 ns	registers and L1 caches **(a)**
main memory	>1000 MB/s	10-100 ns	memory latency **(b)**
SM communication	100-1000 MB/s	10-1000 ns	considering several effects **(c)**
DM communication	100 MB/s	10 μs - 1 ms	dedicated / LAN connections
Local I/O	10-100 MB/s	5 ms	magnetic disks and RAIDs
DM communication	100 MB/s	1-100 ms	WAN, hierarchical **(d)**

(a) Most commercial architectures employ two levels of caching, which we do not explicitly consider here.
(b) Physical memory latency at the chip level is different from, and usually lower than cache miss penalty.
(c) Various mechanisms can influence the performance of shared memory operations: hardware arbitration and synchronization, software locks and thread scheduling. Physical shared-memory has its own caching mechanisms, which are often incorporated into the level-2 processor cache.
(d) While multi-Gigabit geographic networks are being already built, such connections clearly have higher latency, which show the network hierarchic structure.

Hence, in modern parallel architectures we have the following hierarchy of memory and communication layers. Each one of these layers may exhibit hierarchical effects, depending on its implementation choices.

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

Effects of the parallel hierarchy sum to those due to the conventional memory hierarchy of sequential machines. A crucial point is that here is no obvious order or separation among the parameters of these two hierarchies. As we can see in table 1, the bandwidth of some parallel layers is comprised between those of the sequential memory and of the local I/O system, but their latency is usually lower than that of mechanical devices. Assessing the present and future characteristics of this hierarchy [6] and devising appropriate programming models to exploit it are among the main open issues in modern parallel/distributed computing research.

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 toward even more hierarchical architectures than SMP clusters, and we discuss possible future developments.

In [7] some main advantages are found of SMP cluster architectures, most of them being technological and economical considerations.

- Commercial off-the-shelf processors are getting faster and faster, even with respect to special purpose architectures. Because of their quantity, development and production costs are getting lower and lower. Special purpose processors (e.g. vector processors) are no longer able to achieve significant advantages over the commercial product lines, so architectures employing multiple commodity processors are going to be preferred for MPP, SMP and cluster machines.
- A similar effect shows up due to mass-production of network and architecture components. For clusters of small SMP, which employ standard network and structural components, there will be a very fast capability growth. This will lead to cheaper and more scalable networks, which can compete with the special purpose connection structures of massively parallel processing machines (MPP). As soon as the performance advantages of the special purpose networks will disappear, the SMP clusters will get into the position of the MPPs.
- SMP clusters are scalable and expandable. Their architecture is intrinsically more scalable, as we mentioned in section 2, and it is moreover practically expandable by adding more nodes and/or upgrading processing nodes. While it is not possible to add processors in a SMP or MPP, it is easy to build a SMP cluster step by step.

- The size of memory, disk subsystem capacity and bandwidth are a critical resource 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.
- Most software for MPPs or SMPs can be ported to SMP clusters. With the knowledge of software for SMP machines and the already existing software for MPPs, it should be possible to provide a powerful environment for parallel software development and execution. We will give an overview in section 5 of the efforts in this direction.

Some of the preceding considerations have been recognized years ago, while other ones are a more recent discovery. According to [2], the trend will last for more than a while. Bell and Gray depict a scenario of the evolution of parallel computer and computing grid architectures.

Users of supercomputers and proprietary software will turn to proprietary clusters using standard software. The clusters themselves are built from commodity hardware and software. There will be an era of supercomputing mono-culture where every company or research institute will build its own supercomputer.

As a downside, applications which need a large shared memory perform poorly on distributed memory systems. For that reason, computing centers will migrate to super-application centers. They will provide application centric vector- or cellular supercomputers for special research areas.

Computing centers will have the role of fully distributed computation brokers. The centers will decide where, when and on which platform a job is executed, exploiting Grid or metacomputing techniques to manage clusters of supercomputers. Furthermore, computing centers will become super-data centers. They will provide, through a faster Internet, storage for peta-scale data sets with efficient access methods. With the increased Internet bandwidth, cluster and Grid technologies will merge in the next decade. Hence, all LAN-based workstations will become part of clusters, all of them together forming the Grid.

According to this prediction, hierarchical parallel systems will be the principal computing structure in the future. Therefore, investments and research in programming environments and in understanding hierarchical parallelism are very important.

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 [8] as well as 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 based on several precise assumptions on the structure of the machine and the cost of memory accesses.

- An unlimited number of processors (simple processing units with local memories) run the same program. They are connected through an unlimited global memory space where they can read and write in parallel.
- Parallel execution is synchronous, all active processors always complete one instruction in one time unit.
- No bounds are put on the size of the local and global memories, and on the number of active processors.

The PRAM charges the same cost for a computation step, local and global memory access. Practical bandwidth constraints and locality effects are completely ignored. While this assumptions allow to disregard the peculiarities of any specific architecture, and make the PRAM an effective model in studying abstract computational complexity, this set of hypothesis are not realistic for the majority of physical architectures. The PRAM can be seen as simple model of the abstract architecture of fig. 1a, whereas considering the practical issues discussed in section 2 we see that real MIMD machines are much more complex. The PRAM is thus mis-guiding 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, adding different kinds of constraints and costs on the basic operations. A survey on these derived models is given in [9].

In order to account for the details of processor interconnections, part of the research about parallelism concentrated on the study of the communication and algorithmic performance of different network structures. Despite the results about network cross-simulation properties, network-specific algorithms are often too tied to the geometry of the network and show a sub-optimal behavior when running on a different network structure. The design and actual implementation of network-specific programs is difficult, and it heavily depends on the underlying hardware.

A third and different research track has started in the last years, which involves the class of *parallel bridging* computational models. These models, which are the matter of the following sections, aim at exploiting a higher degree of hardware independence. Some of these models explicitly consider the *grain* of the computation as a key factor in devising efficient and practical algorithms.

As early noted by Cormen and Goodrich [1], the bulk-processing nature of external memory algorithms is close to the bulk-parallel approach of most parallel bridging models (PBM). Most practical problems involving large-scale data structures are definitely a target both for parallel computing and for secondary memory computing techniques. They fostered the development of a single computation model which included bulk-like measures for computation, communication and I/O. In this survey we will follow the development path of PBMs with respect to external memory models, as shown in fig. 4, starting from a description of the PBM approach and of the most widely adopted PBMs.

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, useful 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 a single reference model with these characteristics has lead to the concept of *parallel bridging model* (PBM). A PBM should thus 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 in the 90s, which use a more abstract approach in modeling the interconnection architecture.

One of the pioneering works in the field is the paper by Valiant [10], where 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 from 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 is thus optimally simulated on p physical processors if p is rather smaller than v (e.g. $v = \log p$). The approach ensures that there is a 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.

4.1 Bulk Synchronous Parallel and Coarse Grain Multicomputer

The bulk synchronous parallel model (BSP) that Valiant proposes (as described in [11]) is made up of a set of processing nodes with local memories and a complete interconnection network which delivers messages between pairs of nodes (Fig. 2). Three parameters are used to specify the underlying hardware: the

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

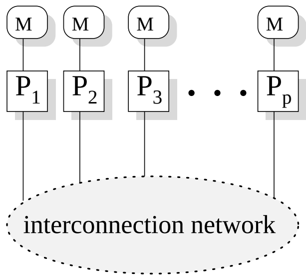

Figure2. (a) BSP symbols and parameters. (b) The BSP abstract architecture.

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 is made 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 are thus a computation phase, a varying routing relation (which expresses the pattern of message exchange) and the constant-bounded synchronization time L (see fig. 2). The table summarizes the essential definitions 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 during current superstep. The routing h-relation among the nodes has size $h_t = \max_i h_{i,t}$ for superstep t. w_t and h_t are the largest w and h values in a superstep. If we imagine that supersteps are globally synchronized by barriers, we can estimate the length of a superstep in time units as $w_t + g \cdot h_t + L$, which becomes instead an upper bound if we assume that synchronization is only enforced when actually needed.

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 algorithm performance 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 $W = c \cdot T_{seq}/p$ and $g \cdot H + L \cdot T = o(T_{seq}/p)$ for a small constant c.

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

The Coarse Grained Multicomputer The Coarse Grained Multicomputer (CGM) model [12] has a similar approach, based on supersteps. CGM supersteps

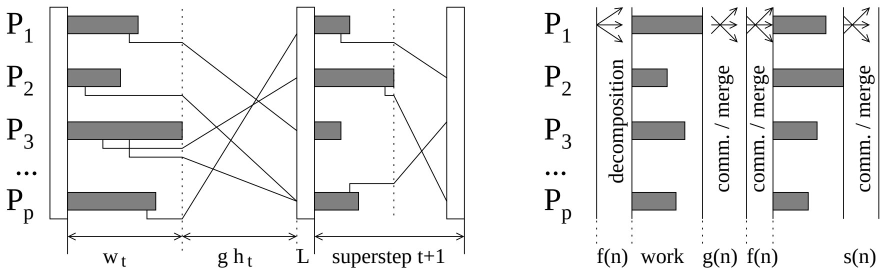

Figure3. (a) BSP superstep execution (b) CGM supersteps

have a more globally fixed nature, thus only two parameters are used, p and n. Each of the p nodes has $O(n/p)$ local memory. The network structure is essentially unspecified in the model, because algorithms exploit a reduced number of fixed, parallel computation patterns (e.g. sorting, broadcasts, partial sums) which can be efficiently implemented over various interconnection networks.

CGM algorithms are given a parametric cost which depends on n, p and the cost of those primitive communication patterns on $O(n)$ data. For each network topology we can then substitute the running time of the specific global operations used (e.g. complexity of exchanging $O(n/p)$ keys in a 2-hypercube of size $\log_2 p$).

A CGM superstep is thus made up of a decomposition phase involving the parallel operations, a local computation phase, and a merge phase also exploiting the parallel primitives. The challenge here is in devising a coarse-grain decomposition of the problem into independent subproblems which exploits 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 become close to BSP. In recent works, CGM algorithms are often defined as a special class of BSP algorithms, which employ at each superstep a routing relation of size $h = \Theta(N/v)$.

LogP model. In the LogP [13] model, processors communicate through point-to-point messages. LogP ignores the network geometry, it models communication behavior using four parameters: l, an upper bound on communication latency, o, the overhead involved in a communication, g, a gap parameter (likewise to the BSP model), and the physical parallelism P. Messages are considered to be of small, fixed length, thus introducing the need to consider partitioning large communications. In the original model (*stalling* LogP) an explicit capacity constraint is imposed on the network, that no processor can have more than $\lceil l/g \rceil$ in transit to it at the same time, and senders in excess to the constraint are assumed to stall.

The unstructured and asynchronous programming model (as well as the capacity constraint requirement) make algorithm design and analysis with LogP more complex than with the other models. There are comparably fewer results with LogP, even if most basic algorithms (broadcasts, summing) have been analyzed in great detail.

QSM model. The Queuing Shared Memory (QSM) [14] model can be seen both as a PRAM evolution and as a shared-memory variants of the BSP. A set of processors with private memory communicate by reading and writing to a shared memory. QSM computation is divided into phases, with reads and writes posted to the shared memory completing only at the end of a phase. Concurrent read (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 read and writes from a same processor, and κ is the *maximum contention* of the phase, i.e. the maximum number of colliding accesses on any location in that phase. The gap parameter g has the usual meaning, whereas latency is not explicitly considered, and it is substituted by the contention. 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 $p \cdot T(N)$ (processors per execution time) on the emulating machine is the same as that on the machine being emulated, to within a constant factor. Since typical emulations increase the amount of parallel slackness (the emulating machine has fewer processor than the emulated one), work-preserving emulations are characterized by a certain slowdown. An ideal slowdown of 1 means that the emulation introduces no inefficiency (within a constant factor). Table 2 summarizes some results on the topic taken from a recent survey by Ramachandran [15]. 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.

On the other hand, 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.

Table2. Slowdown of work-preserving emulation between PBMs. Most of these results concern randomized emulation algorithms. See [15] for details and references.

Emulated model	Emulating 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	$(L/g) + g \log p$	$\log^4 p + (l/g) \log^2 p + g \cdot \log p$	

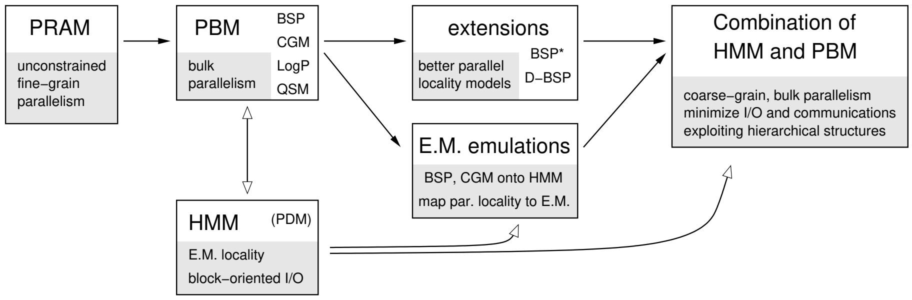

Figure4. Development path of PBMs and their relationship with HMMs

From this point of view, LogP is probably the hardest of the models we have seen, as 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 a 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 more extensively adopted 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. 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, showing up the connections among the design of parallel algorithms and that of external-memory algorithms.

4.2 Parallel Bridging Models and Hierarchical Parallelism

PBMs reward parallel slackness and aggregate computation/communication patterns, so it is not surprising that they are related to external memory models, which reward aggregation of memory accesses. Both kinds of models aim at properly accounting for a kind of access locality.

Two different research paths developed from PBMs (see Fig. 4), which we will survey in this section and the next one. 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 a way similar to some hierarchical memory models. Another path explores the relationships among PBMs and HMMs by developing equivalences and emulation procedures that turn bulk-parallel algorithms into external memory ones.

Because of the initial aim at modeling parallel computation, the main line of development of PBMs is directed toward a more realistic model of parallel locality effects in the computation. As we saw in section 2, 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 quite abstract and simple, and can be made more realistic. We describe two different extensions of BSP, which account in an abstract way for effects due to message length and for the relationship among network size and parallel overhead in communications.

The BSP* model In real interconnection networks communication time is not independent from message length. The combination of bandwidth constraints, startup costs and latency effects is often modeled as an 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. PDM, on the other hand, 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 [11] 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 of message start-ups in a phase (h_t) and for the communication volume s_t (the sum of the sizes of all messages). Superstep cost is defined as $w_t + g^* \cdot (s_t + h_t \cdot B) + L$, where $g^* = g/B$. Each BSP* message is charged a constant overhead which depends on B, and a time proportional to its length. The effect on performance evaluation is that there is no gain in sending messages much shorter than B, and algorithms that pack information when communicating are rewarded. Thus the BSP* model promotes explicitly block-organized communications, as 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 from 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 [16] introduce the decomposable BSP model (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 sub-computations 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 sub-computations.

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 [17]. Meyer auf der Heide and Wanka [11] investigated the relationships among the BSP* and the D-BSP models. Bilardi and others [18] 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 characteristic functions of the form $C \cdot (n/2^i)^\alpha, (0 \leq i \leq \log n) \vee (0 < \alpha < 1)$, which capture a wide family of commonly used interconnection networks, including multidimensional arrays.

4.3 Emulation in External Memory

A classical result of emulation of parallel algorithms using external memory techniques is the PRAM emulation algorithm in [19]. The result is described in [20]. It is an asymptotically good emulation, but the asymptotic complexity hides quite large constant terms, 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 emulation 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 found in [21]. The general scheme of the emulation consists of picking the number v of virtual

BSP processor so as to efficiently simulate them in external memory. The simulation is performed by a sequential external-memory machine, which loads in memory the computational context of each virtual processor (its memory image and message buffers) once for each BSP superstep. Sibeyn and Kaufmann show that simulating the parallel processors one at a time and implementing communications using external memory data structures, we can derive efficient external memory algorithms from a subclass of BSP ones which only require limited memory and communication bandwidth per processor. There are interesting points to remark,

- the use of a *bandwidth gap* parameter, measuring the ratio between the instruction execution speed and the I/O bandwidth,
- the introduction of a notion of x-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 [22] the emulation results hold under more general assumptions. To evaluate the cost of parallel emulation, the EM-BSP* model is defined. EM-BSP* (see fig. 5) is a BSP* model extended with a secondary memory which is local to the processing nodes. 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*, here we find also local memory size M, the number (per processor) of local disks D, the I/O block transfer size B (mutuated 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 [21], the result in [22] exploits the BSP* cost function to simplify the emulation algorithm. BSP*, BSP and CGM algorithms (by reduction to BSP*) can be emulated, which 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) and it is used to compare EM-BSP* algorithms with the best sequential algorithms known.

Parallel and serial external-memory emulation of bulk parallel algorithms is still under development [23,24]. New results focus more on the CGM model, which is seen as a submodel of BSP with a known kind of routing relation (see section 4.1). Despite technical improvements, the overall structure of the

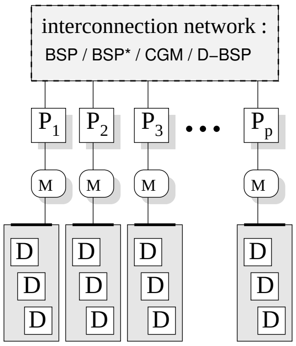

Figure5. The common structure of the combined parallel external-memory models.

model and of the simulation is essentially unchanged from those of fig. 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.

4.4 Algorithms Designed for Parallel Hierarchical Models

In this section we briefly survey two works which exploit mixed models of computation to present a parallel-external memory algorithm. The first work pre-dates most of the results we have previously presented [25]. Aggarwal and Plaxton define a multi-level storage model made up of a set of hypercubes of increasing dimension $0 \leq d \leq a$. A set of four primitive operations is defined on such a structure, which include a scan operation, two routings and a shift of sub-cube data. A parallel hierarchical computation model is then made up of a b-dimensional cube of such devices interconnected along a subset of their elements by bounded-degree networks (it is required that prefix and scan operations are $O(b) = O(\log p)$).

They show how to implement an efficient version of the Sharesort algorithm, and they compare the results to several different hierarchical memory models. Apart from the model and the particular algorithm, it is interesting to note that here it is already defined a parallel, hierarchical data space on which to compute, and the authors choose a set of primitive operation that can be practically implemented both in external memory and in parallel. In a modern setting, this choice allows a certain degree of flexibility in choosing which levels of the computation map to external memory, and which ones to perform in parallel.

The previous work can be fully categorized as a hierarchical model, the next one belongs instead to the track of EM-BSP extensions, thus it only provides two levels of physical memory. However, it uses a D-BSP interconnection structure in the configuration of fig. 5, thus taking into account the hierarchy of the communication network. Since it is nowadays more appealing to solve huge problems by a large cluster of workstation than to switch to tertiary storage, this kind of models is of great practical interest for cluster and grid computing.

In [26] a set of basic primitives for sorting, merging and broadcasting is developed by careful composition of their corresponding D-BSP and PDM ones. Dehne and others develop a distribution sweeping algorithm (see [27]) which works by partitioning the input in parallel and performing local external memory computation. Composing the local and parallel part of the algorithm is not simple, as the sweep paradigm doesn't allow a simple, one-shot input decomposition. Intermediate results of local computations need to be exchanged.

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.

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 (parallel hierarchy, memory hierarchy or both)
- The number of levels (2 or more) that they actually manage.

As we will see, corresponding to the need for a unifying hierarchical model 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 working effort is spent in tuning and debugging activities.

5.1 Tools for Managing Parallelism or External Memory

Software tools in the first level manage only two levels of a hierarchy. Since the mangement of parallelism and I/O is in principle completely separate, these tools can be combined within the same environment to exploit architectures which correspond to the EM - parallel models of sections 4.3, 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 Looking at parallel programming paradigms, we can expose two extremes. First, there is the message passing paradigm. Each processor has its local data and communicates with other processors by exchanging messages. This paradigm represents a *shared nothing* approach. Second, there is the shared-memory programming paradigm. Here all the data can be accessed by all processors. The programmer has to formulate race-conditions to avoid deadlocks or inconsistencies. This is the *shared-everything* approach. The two approaches fit exactly to the two classes SM MIMD and DM MIMD. For both paradigms, there is one de facto standard library. For message-passing-based programming, it is the Message-Passing-Interface (MPI), and for shared-memory programming, it is OpenMP.

Message-Passing-Interface MPI — In 1994, the MPI-Forum unified the most important concepts of message-passing-based programming interfaces. The result of this unification process was the MPI-1 standard [28]. In 1997, MPI-2 [29], an extension to MPI-1, was defined. Within this process, MPI-1 was clarified. A MPI program starts one process per processor on a given number of processors. On each processor the same program is executed, but on different data. Hence, MPI programs have a SPMD (Single Program Multiple Data stream) structure. Each processor receives a rank during the execution. Subject to the rank, a processor can execute different parts of the program. Hence, it is possible to program in MIMD style. The rank is also the address for communication operations. There are MPI-Implementations for nearly all platforms, which is the assumption for portability.

The MPI-1 standard includes mainly the following topics:

- Point-to-point communication
- Process groups
- Process topologies
- Environmental management and inquiry
- Collective operations
- Communication contexts
- Bindings for F77/C
- Profiling interface

The main extensions of MPI-2, besides clarifications of MPI-1 topics, are

- Process creation and management
- Parallel I/O (see MPI-IO in section 5.2)
- External Interfaces
- One-sided Communication
- MPI derived datatypes
- Language bindings for C++

OpenMP — The OpenMP-API [30] is a standard for parallel shared memory programming. OpenMP defines a set of program directives, with which it is possible to define parallel regions in a sequential program. It augments standard C/C++ and Fortran 77/90 directives and facilitates an incremental approach to the parallelization of sequential programs. The base of OpenMP is a fork-join model of parallel execution. The sequential part of the code is executed by one thread (master thread), which forks new threads as soon as a parallel region starts. There are three types of directives:

- **Parallelism/Work sharing** directives define parallel regions in the program, which are executed with multiple threads. All threads perform the same amount of computation. A work sharing directive may appear within the parallel region. It divides 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, which are defined outside a parallel region (for example shared, private and

reduction). Variables defined inside a parallel region are implicitly private to each thread.
– **Synchronization** directives (barrier, critical, flush) are responsible for synchronized execution of the several threads. Synchronization is necessary to avoid deadlocks and data inconsistencies.

In both cases, possible hierarchies in the parallel target machine are not considered. Both paradigms assume independent processors, which are either connected by shared memory or by an interconnection network. Of course, there are approaches to incorporate hierarchical sensitive methods in both libraries. We will present some of them in section 5.3.

External Programming Libraries In parallel programs we can exploit external-memory libraries designed for sequential processing to efficiently access data on processor-local disks. If secondary memory devices are shared among a set of processors, there is the chance that the interaction will lead to a performance loss.

TPIE — TPIE [31] is a library developed as a programming tool to simplify the implementation of algorithms based on the PDM model.

TPIE provides data structures and algorithms to solve batched problems, with a strongly stream-oriented interface to the data. Common operations on streams are provided (e.g. sort, merge, distribution, permutation) as well as on simple external-memory data types (matrices, stacks). The assumption is that all operations and access methods exploit the best known EM algorithm for the problem. Clearly, any TPIE-based program can immediately benefit of theoretical and practical improvements in EM algorithms as soon as they are propagated to the library.

The fundamental components of TPIE are the memory manager and the *block transfer engine* (BTE), the software kernel that moves blocks of data from physical devices to main memory and back. Three different implementations of the BTE are based on the UNIX *stdio*, 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 the PDM algorithm. Extension of the library is planned to the full PDM model, with parallel disks, and to multiple, coordinated BTEs.

LEDA-sm — LEDA [32] is a commercial library of data structures and algorithms for combinatorial and geometric computation. LEDA-sm is a secondary memory extension [33] which is publicly available under the GPL license. It provides external data structures (e.g. arrays, queues, trees, strings) with basic operations, and a few 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.

5.2 Tools for Parallel-External Programming

PDM-like approaches Among the software tools devoted to external memory, the VIC* approach is one that allows to fully exploit the PDM model, with parallel disks and processing elements.

*VIC** — The Vic* compiler [34] is a compiler for the C* language, a C extension with virtual-memory data parallel constructs. The compiler produces code which interfaces to an implementation of the PDM model. 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. PVM or MPI is used for the support of 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 [34].

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 [35], where the problem of out-of-core, parallel FFT is solved by using the dimensional decomposability of the FFT problem to devise a parallel partitioning of the out-of-core computation.

MPI-IO The MPI-2 standard includes the specifcation of a standard parallel I/O API. Programs written using MPI-IO can exploit message passing parallelism and a shared disks 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. 1b,c) trying to access different parts of a single large file. SM architectures often use centralized I/O subsystems, and the target is to minimize contention due this bottleneck. DM architectures usually have local disks in each processing node. In order to exploit these disks as a single storage support, data blocks are sent through the network from hosting nodes to the requesting ones. In both cases the solution to the problem is in aggregating several requests to serve them efficiently. The well-known UNIX-like semantics of most file systems does not allow this operation. The gain is even higher if the program explicitly gives information about collective I/O (parallel, logically synchronized I/O requests from a set o processors).

MPI *derived datatypes* are a portable mechanism to specify the memory layout of a data structure, which 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 [36,37] 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. The papers show that 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, it is allowed 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 a 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 that most of the optimizations which 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. We are thus exploiting the speed relationship among two different kind of hierarchies which we reported in section 2.

5.3 Tools for Parallel-Hierarchical Programming

Hierarchical Optimizations for MPI The advantages of adapting MPI to SMP clusters are mostly in the portability and readability of the code for users. A lot of MPI programs can be ported easily by compilation on the new architectures and performance will be preserved. Further, there is a huge community of MPI developers, whose programming knowledge and experience is not lost. As already mentioned, the message passing paradigm does not consider the hierarchical architecture of SMP clusters. The disadvantage is, that the processors communicate by the network, although they might be in the same node. In the following, we present two approaches for adapting MPI to SMP clusters, which avoid this inefficiency.

Shared-Memory Communication — This approach improves the communication between processors, which reside in the same node by using the faster shared-memory instead of the network for the point-to-point communication. When a message is sent, the system has to detect if the target process works on a processor, which resides in the same node. If this is not the case, the message has to be transferred as usual over the network interface. Otherwise the message will be delivered through the shared memory. The performance of this innernode point-to-point communication depends on the number of copies, which are necessary to transmit the message. In [38] an optimization for a free MPI implementation called MPICH, which works on PC-based SMP clusters, is presented. The version 1.1 was the first version of MPICH, which supported SMP clusters. It needed two shared memory copies to perform the send and receive within the node. The two copies are necessary, because the sending processor writes the message in the shared memory, and the receiving processor reads out of the shared memory to make a local copy. Due to a special network interface (Myrinet), and a special communication software (SCore), it was possible to implement a primitive which reads the message directly out of the senders memory. Hence, only one copy operation is necessary, which increases the point-to-point communication performance.

In order to test the library, the authors made experiments using the NAS Parallel Benchmark 2.3 (NPB) [39]. The NPB is a set of 8 programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications.

The authors compared the results of a SMP cluster with that of an uniprocessor-cluster (UP-cluster), having the same number of processors. The SMP cluster achieved 70-100% of the performance of the UP-clusters. Intuitively, the SMP clusters should perform better, because the shared-memory communication is faster. The following reasons are made responsible that this is not the case.

In general, the latency for each process in a SMP cluster is higher than in an UP-cluster, because multiple processors share the network interface. Since synchronization mechanisms are realized by messages, the costs are dominated by communication costs between the nodes, because most messages have to be sent using the network. If an application has to synchronize a lot, the advantage of the faster innernode communication gets lost. The applications of the NPB are examples of such applications.

Another argument is, that the total number of messages sent by each processor is equal in both cases. Of course some of the messages are sent within the node, but the majority has to be sent through the network. Again, the advantage of the faster innernode communication is exhausted, because of the higher latency for network messages.

Though the point-to-point communication between processors in the same node can be improved by this approach, but it is not guaranteed to improve the overall performance compared to that of an UP-Cluster.

The problem with this approach is, that the programmer is not forced to consider the SMP cluster architecture, which is not helpful for efficiency reasons. He can develop his code as if the underlying machine is a *flat* MPP machine and the efficiency of his program only depends on the MPI implementation and on the general program structure.

Threads Only MPI — Until now, the *thread only MPI* (TOMPI) [40] is just a MPI implementation for single workstations or SMP workstations. The idea is to make the development of MPI programs on workstations less time-consuming. It is inefficient to use the custom MPI implementations, because they use UNIX processes and UNIX domain sockets for inter-process communication, which both involve a large overhead on a single workstation. An implementation using threads and shared memory would perform much faster. TOMPI rewrites an MPI program using a source code translator. The result is a program, using multiple threads on a SMP node or workstation. Messages between processes are copied only once, which is very efficient. With TOMPI it is possible to execute MPI programs with hundreds of MPI processes on a single workstation, without bringing the system down. However, there is no such implementation for SMP clusters now. But, the concept itself could be interesting for SMP clusters. It would be an interesting task to extend the translator towards SMP cluster support. Informations about the structure of the cluster, like for example the number of processors per node or the number of nodes, are necessary as input data.

This approach seems to have the potential to be more efficient than the first one, because it does not only use a faster shared memory communication within the nodes, it even avoids large memory overhead using processes.

Distributed Shared-Memory Programming with OpenMP The usage of OpenMP on SMP clusters brings some advantages along. OpenMP programming is easier than message passing programming. The programmer does not need to take care about data decomposition and distribution. Removing OpenMP directives from a program means to create a sequential version of the program. By converse, it is possible to parallelize a code step by step. The main problem of this approach for SMP clusters is, that there is a need for a global shared memory in physical distributed environment. Software solutions, which provide a distributed shared memory, usually lower the total performance of a program due to memory coherence operations. There are three approaches how to adapt OpenMP to SMP clusters. The first and the second approach use software distributed shared memory (SDSM) systems. In the first approach, OpenMP programs are translated to programs, using library calls to a SDSM systems. The second approach is a compiler directed SDSM approach, where the compiler tries to reduce the overhead of the SDSM system. The third approach does not assume a SDSM system, it is an extension of the set of OpenMP directives, library calls and environment variables for data distribution in a distributed systems.

Software Distributed Shared Memory — SDSM systems provide a global address space for physically distributed memory machines. The idea is to translate OpenMP directives into appropriate calls to a certain SDSM system. 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. An example for this approach is described in [41]. The authors created a translator, which transforms OpenMP programs into standard programs using a special SDSM system, called TreadMarks[42]. Further the SDSM system itself is modified for the use with OpenMP, and for the use on SMP clusters. This modified version of the SDSM exploits the hardware shared memory within a node using POSIX Threads. Experimental testing with several algorithms showed, that the performance of the modified SDSM system got much better compared with the version not using POSIX threads. But, compared with equivalent MPI implementations the performance is still worse. The received speedups were only within 7-30% of the speed-ups achieved by the MPI versions.

Compiler Directed Software Distributed Shared Memory — One problem with the last approach was, that the SDSM system suffers too much from coherence-maintenance network traffic. Further, the SDSM systems do not exploit application specific data access patterns, because communication in shared memory is unknown until run time. A more promising approach is the usage of *compiler directed* SDSM library calls. The OpenMP compiler inserts remote communication primitives to keep the node-distributed memory consistent. This is possible because of the good structure of OpenMP programs. Therefore, the compiler is able to analyze parallel regions in order to optimize communication and synchronization. The design of such a compiler is described in [43]. This approach is based on a runtime DSM library and the compiler analysis of the OpenMP programs. The following extensions are made:

– In a first step, the compiler inserts memory coherent maintenance code of the base DSM runtime system in every access to shared data.
– In a second step, the compiler analyzes the memory access pattern in order to optimize the memory coherence code. Because of the compiler analysis, it is possible to generate efficient collective communication.

In the worst case, when no optimization is possible, the performance should be as good as that of the pure SDSM approach. If optimizations are possible, performance will increase. Regrettably, there are no experimental results in the paper, which could confirm this prediction.

Distributed OpenMP — A different approach to adapt OpenMP to SMP clusters is suggested in [44]. The authors propose the *Distributed OpenMP* (DMP). This extension of OpenMP with data locality features provides a set of new directives, library routines and environment variables. On each node one process is created. There is an one-way communication between the nodes. Each process uses multiple threads according to OpenMP directives and library calls. There is no explicit communication. There are directives, which define the data-distribution. The aim is to maintain full source compatibility with OpenMP. The programmer should start with an OpenMP program and later he can add data mapping directives. At the moment, the approach is not yet available. Hence, the following is just a listing of ideas and issues. Among the proposed extensions, there is a *distribute* directive. With this directive 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 with respect 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, which reside in their local memory. Further extensions are library routines and environment variables, which 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 still has to take care about an efficient data decomposition. Therefore, after adding DMP directives to a OpenMP program, the user probably has to do performance tuning again. The main disadvantage is, that there is no clear idea nor real result the feasibility of this approach.

An advantage of all three approaches is, that one can have a single portable code for SMP systems, distributed memory systems or SMP clusters. Further, the performance of one SMP node will not degrade, and the code can still be parallelized incrementally.

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. The following section will 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, which compare hybrid programs with pure MPI ones. The general execution scheme is as follows. In each node there is one MPI process. Communication between the nodes is done by the MPI processes. Inside the process, multiple threads are responsible for doing the parallel computation. The number of threads, spawned within 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 [45], there are two approaches to incorporate OpenMP directives into MPI programs. The fine-grain and the coarse-grain approach.

Fine-Grain Parallelization — This is the straightforward approach, because it can be done incrementally. The computational part of the MPI code is examined, and the loop nests are parallelized with OpenMP directives. Therefore, the approach is also called loop-level parallelization. In order to avoid an unnecessary increase in programming effort, the loop-nests of a program must be profiled according to their contribution to the global execution time. Only loop nests with a significant contribution are selected for OpenMP parallelization. Some loop-nests can not be parallelized directly. But if they contribute significantly to the global execution time, the developer can try to transform these loop-nests into parallel ones, to avoid false sharing or to reduce the number of synchronizations. Techniques for parallelizing non parallel loop-nests are loop permutations or exchanges and the use of temporary variables.

Coarse-Grain Parallelization — In this approach a SPMD programming style is used to incorporate OpenMP into MPI programs. OpenMP is used to spawn threads immediately after the spawn and initialization of the MPI processes in the main program. Each thread itself is acting similar to a MPI process. There are several issues which have to be considered for the threads.

- 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, until now, there has been no work about the coarse-grain approach. Remembering the idea of *TOMPI*, it seems, that the result of TOMPI for SMP clusters is a coarse-grain hybrid program, because the MPI processes are converted to threads. An important thing, which has to be incorporated in an TOMPI for SMP clusters is the usage of common data structures within the nodes as proposed by the coarse-grain approach.

Performance of Fine-Grain Hybrid Programs compared to Pure MPI Programs — Beside knowing, how to develop hybrid programs for SMP clusters, it is necessary to look at the achieved performance of the programs. In [45,46,47,48] investigations to measure performance of fine-grain hybrid programs are presented. They try to measure, which programming model performs better on a SMP cluster by comparing the performance achieved by a hybrid and a pure MPI version of NPB. Further, they investigate if it is worth building a SMP cluster instead of an UP-cluster, and they tested, if a PC-based SMP cluster can compete with other supercomputers. An important subject of the papers is the interpretation of the performance measurements. The authors try to explain the behavior of the various programs on SMP clusters in order to understand performance for hybrid programs. 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.

According to the mentioned works, the comparison among the two kinds of models for SMP clusters show 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 are responsible, which model performs better.

- **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**. The communication time depends on the communication pattern of an application. To be more precise it depends 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 the data fast enough to the network interface to achieve the maximum bandwidth of the network. Therefore, the pure MPI approach performs better, if the application is bandwidth limited. Otherwise, for latency limited application the pure MPI approach is worse.
- **Memory access patterns**. The memory access patterns are different for the two models. Whereas MPI allows to express multi-dimensional blocking, it is not natural for OpenMP to do so. 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)**. If the processors are so fast, that communication becomes the bottleneck, then the communication pattern decides which model is best. Otherwise, if computation is the most significant part, then MPI seems to be always the best.

Parallel Radix-Sort – An example for improving an algorithm by using hybrid programming is *parallel Radix-Sort*. In [49] there is a presentation of the transition from a message-passing parallel Radix-Sort algorithm to a hybrid parallel Radix-Sort algorithm using POSIX threads and a special internode communication library. The hybrid implementation was compared with a pure MPI implementation and a DSM implementation on a cluster of DEC AlphaServer 2100. All experiments confirmed, that the hybrid implementation has the best performance. In the following, we will show the design of the hybrid parallel Radix-Sort algorithm. As mentioned in section 5.4, we start with the message-passing version, then we replace multiple MPI processes per node with a single process and multiple shared-memory threads, and finally, we parallelize sequential work into shared-memory algorithms.

We consider the problem of sorting n integers keys in the range $[0, M-1]$ distributed evenly over the SMP cluster. The cluster consists of p SMP nodes, where each has r processors. The integer keys are divided into groups of d digits. Radix-Sort sorts the keys using a Counting-Sort algorithm on each digit group. We start with the least significant bit. Let $M = 2^b$, then we need $\frac{b}{d}$ passes of Counting-Sort. In each pass, the integer keys are sorted by one d-digit group starting from the least significant group. The Counting-Sort algorithm sorts n integers in the range $[0, 2^d - 1]$ by using 2^d counters. The counters are used to accumulate the number of keys with value i in bucket B_i $(0 \leq i < 2^d)$. After counting the numbers of the keys, the rank of each key can be calculated. Once the ranks are known, we can move the keys into their correct position. This can be done by a collective communication and we assume, that no node is the source or destination of more than $\frac{n}{p}$ keys. The pseudo code for Counting-Sort consists of six steps.

1. Each node i, $0 \le i < p-1$, counts the quantity of each value for all its $\frac{n}{p}$ keys and computes the array $H_i(k)$, which is the number of keys equal to k $(0 \le k < 2^d)$ in node i.
2. Each node applies an all-to-all communication on its H array with block size $2^d/p$. After this step, each node has p rows of size $2^d/p$.
3. Each node locally computes the prefix-sums of its rows of the array H for each key. Further, the node builds an array with the total count for each key.
4. Each node applies an all-to-all communication on the computed prefix-sums and the total count array. The block size is $2(2^d/p)$.
5. With the received arrays, each node can compute the ranks of its $\frac{n}{p}$ local keys.
6. Each node performs a collective communication of keys to rank locations, whereby no node is the source or destination of more than $\frac{n}{p}$ keys.

The computation steps can be parallelized within each node as follows. The communication steps (2,4,6) remain unchanged. The modified steps and the unmodified steps form the hybrid program.

- In step 1, each processor of the node scans in parallel $\frac{n}{rp}$ keys and creates an array with the quantity of each value. After that the r arrays can be merged in parallel into a single array for the node.
- In step 3, the $\frac{2^d}{p}$ prefix-sums can be computed independently. The work can be distributed evenly among the r threads. Each thread computes $\frac{2^d}{rp}$ prefix-sums.
- In step 5, the computation of the ranks can be done in parallel by the r threads. Each thread computes $\frac{n}{rp}$ ranks of the local keys.

Basically, the pseudo code shows the design of a parallel hierarchical hybrid program. The single steps of the shared memory computation may be subject to further optimizations concerning the innernode memory hierarchy.

5.4 Models for Exploiting Hierarchical SMP Clusters

Besides the programming libraries and paradigms above, there are some programming models for SMP clusters, which try to build a higher level of abstraction for the programmer.

SIMPLE Model The SIMPLE model [49] efficiently combines shared memory programming within a node with message passing between the nodes. The target parallel architecture of the model consists of a collection of SMP nodes interconnected by a network. The network can be modeled as a complete graph, hence, communication is determined by latency and bandwidth properties of the network. Each node contains several processors which have uniform access to the shared memory and other resources like for example the network interface. A parallel algorithm is defined as a sequence of local (parallel) computation, interleaved with communication steps. This resembles BSP-algorithms [10]. The programming of the SMP-nodes is based on threads which communicate via synchronized access to the shared memory. The model does not restrict that all nodes must be identical. Therefore, the number of threads per node is not available globally. Because of that, the model provides only node-oriented communication. At any given time, only one thread of a source node can send, and only one thread of the a target node can receive a message. SIMPLE provides several primitives, which help to design efficient algorithms. These primitives can be divided into two categories, communication and computation. The computation primitives comprise data parallel loops, control primitives to address threads or nodes directly, and memory managment primitives. SIMPLE provides three libraries for communication. There is an internode-communication library based on MPI, a SMP node library for the thread synchronization and a SIMPLE communication library build on top of both.

Together with the programming model, SIMPLE provides a method to design efficient algorithms on the base of message-passing algorithms. The most costly operations are the internode communication operations. Compared to access times to local memory, latency of communication over the network is an order of magnitude higher. Therefore, designing an efficient algorithm means minimizing the internode communication operations. Since, this is similar to the design of efficient message passing algorithm, a possible approach is, first, to design an efficient message passing algorithm for the SMP cluster, only considering the computing nodes. Second, turning it into an efficient SIMPLE algorithm. This can be done by mapping the computational work assigned to a node into an efficient SMP algorithm. The result of this process is an efficient SMP cluster algorithm. An example for such a method is the parallel Radix-Sort algorithm, which was presented in the previous section.

KeLP2 Model Kelp2 [50] is a runtime infrastructure, that implements a methodology for implementing block-structured numerical applications on SMP-clusters. KeLP2 provides mechanisms to coordinate data decomposition, data motion and parallel control flow. Beside that, it hides low-level details like message-passing, processes, threads, synchronization and memory allocation from the programmer. There is no analysis of the code in order to make high-level restructuring and there is no automated data decomposition. But, the model empowers the user to make such decisions. The programmer can do the decomposition best and KeLP2 provides a framework to construct partitioning libraries. KeLP2 supports three levels of control.

1. collective level (SMP-cluster)
2. node-level
3. processor-level

Programs express parallelism at the node- and processor-level while communication takes place in the collective- and node-level. As in the SIMPLE model, KeLP2 assumes that communication between two processors of different nodes can only be done through their nodes. Each level, the collective- and the node-level, has its own data-layout and -motion. The KeLP2 programming abstractions help to manage this. There are three classes of KeLP2 programming abstractions.

1. Meta-Data
2. Instantiators
3. Parallel Control Flow

The meta-data represents the abstract structure of some facet of the calculation. It describes the data decomposition and the communication patterns. Instatiators execute the program according to the informations contained in the meta-data. The primitives for parallel control flow are iterators, which iterate over all nodes or over all processors of a specified node. KeLP2 enables a parallel specification, that is less dependant from the implementation.

6 Conclusions

It is clear from section 2 hat we need theoretical models and new software tools to fully exploit hierarchical architectures like large clusters of SMP and future Computing Grid super-clusters.

The interaction among parallel bridging models and external memory ones has produced several results which we surveyed in sections 3 and 4. The exploitation of locality effects in these two classes of models employs very close solution, which involve block-oriented cooperation and abstract modeling of the hierarchical structure,

The concept underlying theoretical and performance results of 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 way to go in order to answer the need of appropriate computational models. Hierarchical-parallel models like those of section 4.3 and 4.4 are already close to the structure of modern SMP clusters, and they are relatively simple to understand. We can often appropriately compose BSP and PDM algorithms to develop efficient solutions, but the analysis task is still complex.

Models like the one in fig. 5 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 7, 8 parameters. Excessive complexity of the analysis and poor intuitive understanding are a limiting factor for the diffusion of a computation models, as it was pointed out in [1]. There is no answer yet to the questions "can all these parameters be merged in some synthesis?", what are the most four or five important parameters?

Simple disk I/O models like PDM have been exploited to evaluate a corpus of external memory algorithms that are widely used in practice. On the other hand, the impact of more sophisticated parallel computational model is still limited. Most of the efforts in software developing are spent in maintaining existing programming interfaces and optimizing them for the new architectures.

This approach has not yet been proved effective for hierarchical parallel architectures. 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 for the compilation and support tools to achieve implementations with optimal performance.

Most standard libraries only exploit one level of memory or parallel structure, like MPI, OpenMP, TPIE. While it is possible to compose libraries which manage well-separate hierarchies (e.g. disks and shared memory), we have seen in section 5 that the hybrid programming model, resulting from the empirical combination of MPI and OpenMP, leads to much harder problems.

The hybrid model is the only parallel and hierarchical programming model accepted in the practice. It has the driving advantage of using existing, available tools, but has numerous drawbacks: 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 of the structure and the amount of hidden interactions among the architecture, the algorithm and the different software tools exploited, make it quite difficult to devise performance models for the resulting programs.

Two research directions are thus still open. One is to devise parallel hierarchical models which are sound and intuitive, and that 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 works in the literature, an important resource is a set of common, basic operation which are efficiently performed both in the parallel hierarchy and in the memory one. 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 the software tools. Abstract, high level operations simplify program writing, while still providing information for the hierarchy mapping. The SIMPLE and KeLP2 systems are two solutions close to this research path.

A different kind of research, which is going on at the same time, aims at simplifying the management of hybrid parallel programming. Code translators (e.g. from MPI to hybrid structured code), and semi-automatic restructuring tools, look at the moment two feasible solutions for easing and speeding up hybrid software development, which is too slow and fault-prone.

References

1. Thomas H. Cormen and Michael T. Goodrich. A bridging model for parallel computation, communication, and I/O. *ACM Computing Surveys*, 28(4es):208–208,

December 1996. Position Statement.

2. Gordon Bell and Jim Gray. High performance computing: Crays, clusters, and centers. what next? Technical Report MSR-TR-2001-76, Microsoft Research, Microsoft Corporation, August 2001.
3. http://www.top500.org, 2001.
4. Aad J. van der Steen and J. Dongarra. Overview of recent supercomputers, November 2001.
5. Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Pub., July 1998.
6. Andrew A. Chien. Computing platforms. In Foster and Kesselman [5], chapter 17.
7. William E. Johnston. Rationale and strategy for a 21st century scientific computing architecture: The case for using commercial symmetric multiprocessors as supercomputers. *International Journal of High Speed Computing*, June 1998.
8. Jeffrey Scott Vitter. External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001.
9. A. Grama, V. Kumar, S. Ranka, and V. Singh. Architecture independent analysis of parallel programs. In Alexandrov et al. [51], pages 599–608.
10. Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
11. F. Meyer auf der Heide and Rolf Wanka. Parallel bridging models and their impact on algorithm design. In Alexandrov et al. [51], pages 628–637.
12. F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse-grained multicomputers. In *Proc. of the 9th Symp. on Computational Geometry*, pages 298–307. ACM, 1993.
13. D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Comm. ACM*, 39(11):78–85, 1996.
14. Philip B. Gibbons, Yossi Mathias, and Vijaia Ramachandran. Can shared-memory model serve as a bridging model for parallel computation? In *Proc. of the 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 72–83, 1997.
15. Vijaya Ramachandran. Parallel algorithm design with coarse-grained synchronization. In Alexandrov et al. [51], pages 619–627.
16. P. De La Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 - Parallel Processing, vol. II*, volume 1124 of *Lecture Notes in Computer Science*, pages 352–360, 1996.
17. Martin Beran. Decomposable bulk synchronous parallel computers. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99: Theory and Practice of Informatics*, volume 1725 of *Lecture Notes in Computer Science*, pages 349–359, 1999.
18. G. Bilardi, C. Fantozzi, A. Pietracaprina, and G. Pucci. On the effectiveness of D-BSP as a bridging model of parallel computation. In Alexandrov et al. [51], pages 579–588.
19. Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roverto Tamasia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proc. of the 6th Annual Symposium on Discrete Algorithms*, pages 139–149. ACM, 1995.
20. Anil Maheshwari and Norbert Zeh. A survey of techniques for designing.... In *THIS BOOK*. Springer, 2002.
21. Jop F. Sibeyn and M. Kaufmann. Bsp-like external memory computation. In *Proc. 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 229–240, 1997.

22. Frank Dehne, Wolfgang Dittrich, and David Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *SPAA '97*, pages 106–115, 1997.
23. F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Reducing i/o complexity by simulating coarse grained parallel algorithms. In *Proc. 13th International Parallel Processing Symposium (IPPS'99)*, pages 14–20, 1999.
24. F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory of Computing Systems*. To appear.
25. Alok Aggarwal and C. Greg Plaxton. Optimal parallel sorting in multi-level storage. In *Proc. Fifth Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 659–668, 1994.
26. Frank Dehne, Stefano Mardegan, Andrea Pietracaprina, and Giuseppe Prencipe. Distribution sweeping on clustered machines with hierarchical memories. In *Proc. of 2002 IPDPS Conference*, apr 2002.
27. xxx yyy. chapter about comp. geometry algorithms. In *THIS BOOK*. Springer, 2002.
28. Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report CS-94-230, Computer Science Department, University of Tennessee, Knoxville, TN, May 1994.
29. Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, July 1997.
30. OpenMP Forum. OpenMP C and C++ application program interface, version 1.0. http://www.openmp.org, October 1998.
31. Lars Arge, Rakesh Barve, David Hutchinson, Octavian Procopinc, Laura Toma, Darren Erik Vengroff, and Rajiv Wickeremesinghe. *TPIE User Manual and Reference*, 0.9.01b edition, November 1999. Draft.
32. S. Näher and K. Mehlhorn. LEDA: A platform for combinatorial and geometric computing. *Communications of the ACM*, 1995.
33. Andreas Crauser and Kurt Mehlhorn. *LEDA-SM, A Platform for Secondary Menmory Computation*. Max-Planck-Institut für Informatik, Saarbrücken, Germany, March 1999.
34. Thomas H. Cormen and Melissa Hirschl. Early Experiences in Evaluating the Parallel Disk Model with the ViC* Implementation. Technical Report PCS-TR96-293, Dartmouth College, Computer Science, Hanover, NH, September 1996.
35. Lauren M. Baptist and Thomas H. Cormen. Multidimensional, multiprocessor, out-of-core FFTs with distributed memory and parallel storage. In *11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 242–250, June 1999.
36. Rajeev Thakur, William Gropp, and Edwing Lusk. A case for using mpi's derived datatypes to improve i/o performance. In *Proc. of SC98: High Performance Networking and Computing*. IEEE, November 1998.
37. Rajeev Thakur, William Gropp, and Edwing Lusk. On implementing mpi-io portably and with high performance. In *Proc. of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM, May 1999.
38. Toshiyuki Takahashi, Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Horoshi Harada, Yutaka Ishikawa, and Peter H. Beckman. Implementation and evaluation of MPI on an SMP cluster. In *IPPS/SPDP Workshops*, pages 1178–1192, 1999.

39. David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.
40. E. Demaine. A threads-only MPI implementation for the development of parallel programs. In *11th International Symposium on High Performance Computing Systems, HPCS*, pages 153–163, 1997.
41. Y. C. Hu, H. Lu, A. Cox, and W. Zwaenepoel. OpenMP for networks of SMPs. In *Proc. of the Second Merged Symp. IPPS/SPDP 1999)*, 1999.
42. C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.
43. M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster, 1999.
44. J. Merlin, D. Miles, and Vincent Schuster. The portland group distributed OMP: Extensions to openMP for SMP-cluster. EWOMP 2000, September 2000.
45. F. Cappello and D. Etiemble. MPI versus MPI+openmp on IBM SP for the NAS benchmarks. In *Supercomputing*, 2000.
46. F. Cappello and O. Richard. Performance characteristics of a network of commodity multiprocessors for the NAS benchmarks using a hybrid memory model. In *IFIP PACT 99*, October 1999.
47. F. Cappello, O. Richard, and D. Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *IEEE HPCA6*, January 2000.
48. E. Chow and D. Hysom. Assessing performance of hybrid MPI/openMP programs on SMP clusters. Technical Report UCRL-JC-143957, Lawrence Livermore National Laboratory, May 2001.
49. D.A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.
50. S. B. Baden and S. J. Fink. A programming methodology for dual-tier multicomputers. *IEEE Transactions on Software Engineering*, 1998.
51. V. N. Alexandrov et al., editors. *ICCS 2001*, volume 2074 of *Lecture Notes in Computer Science*, 2001.