Designing Parallel Algorithms for SMP Clusters

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Martin Schmollinger
aus Reutlingen

Tübingen
2003

Tag der mündlichen Qualifikation:	29.10.2003
Dekan:	Prof. Dr. Martin Hautzinger
1. Berichterstatter:	Prof. Dr. Michael Kaufmann
2. Berichterstatter:	Prof. Dr. Wolfgang Rosenstiel

Danksagungen

Es gibt Momente im Leben, in denen man wichtige und richtungweisende Entscheidungen treffen muss. Man sollte sich glücklich schätzen, wenn man rechtzeitig bemerkt, dass ein solcher Moment vorliegt. Ich möchte all denen danken, die mich ermuntert haben, mir die Zeit für die Promotion zu nehmen.

Vor allem möchte ich meinem Betreuer Prof. Dr. Michael Kaufmann danken, der mir die Möglichkeit gegeben hat diese Arbeit anzufertigen und mich dabei in allen Belangen unterstützt hat.

Bedanken möchte ich mich auch bei Prof. Dr. Wolfgang Rosenstiel für seine Arbeit als zweiter Berichterstatter.

Mein Dank gilt auch meinen aktuellen und früheren Kollegen des Arbeitsbereichs Paralleles Rechnen für die gute Atmosphäre und die vielen Unternehmungen wie Betriebssport, "Besseressen" oder Boule.

Zu guter Letzt bedanke ich mich auch bei meinen Freunden und Eltern, die im Gegensatz zu mir immer an den erfolgreichen Abschluss der Promotion geglaubt haben.

Zusammenfassung

In der vorliegenden Dissertation untersuchen wir Entwurfs- und Optimierungsmethoden für die Entwicklung von parallelen Algorithmen für *SMP Cluster*. Dabei handelt es sich um eine spezielle Architektur von Parallelrechnern, die zwei verschiedene Konzepte in einem System vereint. SMP Cluster bestehen aus Rechenknoten, deren Prozessoren speichergekoppelt sind (shared-memory). D.h. die Prozessoren innerhalb eines Rechenknotens können über den gemeinsamen Speicher kommunizieren und synchronisieren. Die Rechenknoten selbst werden durch ein Verbindungsnetzwerk miteinander verknüpft. Die Kommunikation und Synchronisation von Prozessoren in verschiedenen Rechenknoten erfolgt über dieses Netzwerk und entspricht einem nachrichtengekoppeltem System bzw. einem System mit verteiltem Speicher (distributed-memory). Diese Organisation führt zum einen zu einer parallelen Hierarchie, denn Parallelität gibt es sowohl innerhalb als auch zwischen den Rechenknoten. Zum anderen entsteht eine Hierarchie bezüglich der Kommunikation. Im Allgemeinen ist die Kommunikation innerhalb eines Rechenknotens bei der ein gemeinsamer Speicher verwendet wird schneller, als die Kommunikation über ein Verbindungsnetzwerk. Es existieren demnach mindestens zwei Hierarchiestufen. Durch moderne Entwicklungen wie hierarchische Strukturen von Verbindungsnetzwerken, Metacomputing Technologien, bei der mehrere Parallelrechner verbunden werden oder Grid Computing Technologien, die das Internet verwenden um weltweit verteilte Rechenressourcen zu vereinen, kann es jedoch weitere Hierarchiestufen geben. Auch hier gilt, je "niedriger" die Ebene in der Netzwerkhierarchie, desto schneller ist die Kommunikation.

Aus diesem Grund müssen effiziente Algorithmen in der Art gestaltet werden, dass sowohl die parallele Hierarchie als auch die Kommunikationshierarchie berücksichtigt werden. Im Allgemeinen werden beim Entwurf von Algorithmen die Hierarchien vernachlässigt und die Maschine als nicht hierarchisch angesehen. Darüber hinaus ist die allgemeine Vorgehensweise bei der Erstellung von Programmen dominiert durch das jeweils verwendete Programmiermodell. Der Entwickler verlässt sich häufig auf die Effizienz der verwendeten Bibliotheken. Die Verwendung optimierter Bibliotheken führt sicherlich ebenfalls zu sehr schnellen Program-

men, doch um wirklich das Optimum zu erreichen ist es nötig auch die Algorithmen an die hierarchische Umgebung anzupassen. Ein weiteres Problem ist, dass der Entwurfprozess paralleler Algorithmen in der Regel nicht gestützt ist durch die Verwendung von Methoden. Sicherlich kann die Entwicklung eines parallelen Algorithmus nicht auf ein einfaches Rezept reduziert werden, dennoch kann die Verwendung allgemeiner Methoden die Menge berücksichtigter Alternativen erhöhen und fehlerhafte Ansätze vermeiden.

In den folgenden Kapiteln zeigen wir einen alternativen Weg auf, der die Zusammenhänge zwischen theoretischen Kostenmodellen, Programmiermodellen und SMP Clustern aufzeigt und es dadurch ermöglicht eine theoretische Analyse eines Algorithmus in eine effiziente Implementierung für SMP Cluster münden zu lassen. Neben dieser Brücke von einer theoretischen Analyse zur effizienten Implementierung stellen wir verschiedene Methoden für den Entwurf und die Optimierung von parallelen Algorithmen für SMP Cluster vor. Anhand verschiedener Fallbeispiele erklären wir die Methoden und die Verwendung des Kostenmodells bei der Analyse der entwickelten Algorithmen.

Kapitel 1 ist eine Einführung in die Problematik der Entwicklung von parallelen Algorithmen. Neben einem motivierenden Beispiel wird der Zusammenhang zwischen theoretischen Kostenmodellen, Programmiermodellen und Architekturen dargestellt.

Kapitel 2 gibt dann eine Übersicht über parallele Architekturen, Kosten- und Programmiermodelle und zeigt ihre mögliche Verwendung für den Entwurf von Algorithmen für SMP Cluster auf.

Wir beginnen mit der Formulierung eines theoretischen Kostenmodells für SMP Cluster (κNUMA) in Kapitel 3 und zeigen seine Verwendung anhand der Analyse von *broadcast* Problemen, bei denen ein Prozessor eine (individuelle oder einheitliche) Nachricht an jeden anderen Prozessor übermitteln muss. Wir betrachten das vorgestellte Modell als eine Obermenge für die Analyse von Algorithmen auf SMP Clusters. In Abhängigkeit der Eigenschaften eines speziellen SMP Cluster und aufgrund der Beschaffenheit des zu untersuchenden Problems kann die Verwendung eines reduzierten Modells ausreichend sein und macht die Analyse leichter.

Kapitel 4 stellt Methoden zur Entwicklung von parallelen Algorithmen und zur Optimierung für SMP Cluster vor. Die Methoden können auf den verschiedenen Ebenen des Entwicklungsprozesses angewendet werden, beginnend bei einer fein-granularen Zerlegung des gegebenen Problems bis hin zur Optimierung einzelner Aspekte paralleler Algorithmen für SMP Cluster.

Die weiteren Kapitel sind Fallstudien für die Verwendung der Methoden bei der Algorithmusentwicklung und die Verwendung des Kostenmodells bei der Analyse.

In Kapitel 5 zeigen wir anhand der parallelen Matrix-Vektor Multiplikation, wie ein paralleler Algorithmus auf die SMP Cluster Architektur übertragen wird, wie redundante Daten genutzt werden können um Kommunikationsoperationen einzusparen und wie eine unnötige Verwendung von redundanten Daten vermieden wird. Dazu werden verschiedene Datenverteilungen untersucht.

Kapitel 6 stellt anhand des Problems der Transponierung von verteilten Matrizen in paralleler Umgebung eine Optimierungsmethode vor, die versucht existierende Kommunikationsmuster durch geschickte Datenverteilung an die jeweilige Struktur des SMP Clusters anzupassen um die Kommunikationskosten zu reduzieren. Wir stellen eine Verteilung vor, durch die das Transponieren einer verteilten Matrix keine Kommunikation über das Netzwerk benötigt und gleichzeitig den Speicherbedarf je Prozessor minimiert.

Kapitel 7 ist eine Fallstudie für den Entwurf eines hierarchisch-sensitiven Algorithmus. Die Methode schlägt vor ein Problem durch Informationsaustausch in immer niedrigere Ebenen der Hierarchie zu verschieben, bis am Ende lediglich lokale Berechnungen nötig sind. Als Beispiel wird das *Radix Sort* Verfahren untersucht. Dieses Verfahren ermöglicht es ganze Zahlen anhand ihrer Binärdarstellung in mehreren Iterationen zu sortieren. Dabei werden in jeder Iteration die Zahlen anhand eines Teils ihrer binären Darstellung in eine Reihenfolge gebracht. Wir zeigen den Weg von der sequentiellen Version bis zu einer hierarchisch sensitiven parallelen Version auf, die mit einem Minimum an Kommunikation auskommt und daher exzellent für SMP Cluster geeignet ist.

Die Ergebnisse der Arbeit sind in Kapitel 8 zusammengefasst. Der Anhang A gibt eine Übersicht der begutachteten Publikationen, die zu den einzelnen Kapiteln veröffentlicht wurden.

Preface

In the following thesis, we observe methods for designing and optimizing parallel algorithms for *SMP clusters*. This particular architecture for parallel computers combines two different concepts. SMP cluster consist of computing nodes that are shared-memory systems, because the processors have access to common resources and especially to the local memory system. Hence, the processors within the same node are capable to communicate and synchronize using the shared-memory. An interconnection network connects the nodes. Communication and synchronization of processors from different nodes is done over this network and thus, correspond to a distributed memory system. In the first place, this organization leads to a parallel hierarchy, because parallelism is involved within and between the nodes. Secondly, a hierarchy is created concerning communication. In general, communication within a node is faster than communication between the nodes due to the use of shared-memory. Therefore, there are at least two levels of hierarchy. Due to modern trends like hierarchical interconnection structures, Metacomputing technology, where several parallel machines are connected, or Grid computing technology that use the Internet to unify distributed computing resources in the whole world, there might be even more levels of hierarchy. Basically, the lower the level of the network for a communication operation, the faster the communication can be done.

On this account, efficient algorithms have to be designed in the way that the parallel as well as the communication hierarchy is considered. Usually, this is not the case, and the SMP clusters are regarded as *non-hierarchic*. Moreover, the general approach for the design of a parallel application is dominated by the use of a particular programming model. The programmer relies on the efficiency of the implementation of the model for the respective platform. Of course, this strategy does also lead to fast programs, but in order to reach the optimum, it is additionally important to adapt the algorithms to the hierarchical environment. Another problem is that the design process of parallel algorithms is in general not supported by methods. Obviously, the design of a parallel algorithm cannot be reduced to a simple recipe, however, the consideration of general design methods maximizes the amount of considered options and minimizes the threat of wrong algorithm design approaches.

In the following chapters, we show an alternative way that shows the dependencies between theoretical cost models, programming models and SMP clusters. It enables the developer to convert a theoretical analysis of an algorithm into an efficient implementation for SMP clusters. Besides the bridge from a theoretical analysis to an efficient implementation, we present several methods for designing and optimizing parallel algorithms for SMP clusters. With the help of case studies, we explain the design methods and the usage of the cost model for the algorithm analysis.

Chapter 1 is an introduction to the issues of developing parallel algorithms. Besides a motivating example for parallel algorithms, the connections between theoretical cost models, programming models and parallel architectures are depicted.

Chapter 2 gives an overview on parallel architectures, cost- and programming models and shows their capability for the development of parallel algorithms for SMP clusters.

After that, we start with the formulation of a theoretical cost model for SMP clusters (κNUMA) in Chapter 3 and show its usage by the analysis of *broadcast* problems, where one processor has to send one (individual or general) message to each of the other processors. The model can be regarded as a super-set for the analysis of algorithms for SMP clusters. Depending on the properties of a certain SMP cluster and because of the character of the analyzed problem, the usage of a reduced model may be sufficient and makes the analysis more feasible.

Chapter 4 introduces methods for designing and optimizing parallel algorithms for SMP clusters. The methods can be applied to different stages of the design process, beginning at the stage of a fine-grained problem partitioning and ending with the optimization of single aspects of parallel algorithms for SMP clusters.

Further chapters are case studies for the usage of the methods for algorithm design and for the usage of the cost model for the analysis.

In Chapter 5, we show by means of the parallel dense matrix-vector-multiplication how a parallel algorithm is transferred to an SMP cluster, how redundant data can be used to reduce the number of communication operations and how an unnecessary usage of redundant data can be avoided. For that purpose, we analyze several data-distributions.

Chapter 6 presents an optimization method that tries to adapt existing communication patterns to the respective structure of an SMP cluster. Communication patterns of algorithms can often be influenced by data distribution. We attempt to distribute the data in order to reduce the communication cost maximally. The method is explained using the problem of transposing distributed matrices in a parallel setting. We present a data distribution with which an algorithm is able to transpose a distributed matrix without communication over the network. At the same time, the data-distribution minimizes the amount of memory per processor.

Chapter 7 is a case study for the design of a hierarchic-sensitive algorithm. The method suggests moving a problem in lower and lower levels of the hierarchy by exchanging informations, until only local computation remains. As an example, the *Radix Sort* algorithm is observed. This method enables to sort integer values by their binary representation within several iterations. In each iteration, the integers are sorted according to a certain part of their binary representation (the radix). We show the way from a sequential to hierarchical sensitive parallel algorithm that works with a minimum of communication operations and hence, is very suitable for SMP clusters.

The results of the thesis are summarized in Chapter 8. The Appendix A gives an overview on refereed publications that build the base for the chapters.

Contents

Chapter 1

Introduction

1.1 Motivation for Parallel Computation

Parallel computing is based on the simple real world observation that generally several workers can finish a job much faster than only one. The *speed-up* that can be achieved by parallelism mainly depends on the structure of the job. If we consider a set of products, which has to be carried from a warehouse into a van, then it is clear that several workers can accelerate this job optimally by their number. In the following analysis of the example, we neglect the arithmetic precision and assume that the products can be assigned evenly to the workers. If we have n products and p workers, then every worker carries n/p products to the van. Since each worker has to do the same amount of work, we can say that the work is balanced in an optimal manner. In computer science, the problem of distributing the work evenly is called *load balancing*. If t is the time a worker needs to carry one product to the van, then the total time for loading the van is tn/p. If only one worker ($p = 1$) loads the van, then the total time is tn. In computer science, the *speed-up* of a parallel algorithm is defined as the quotient between the running time of the fastest sequential algorithm for a problem and the running time of the parallel algorithm. Hence, if we use p workers then the job has a speed-up of $tn/(tn/p) = p$. This can be regarded as optimal, because we invest p workers and the job runs p times faster, see Fig. 1.1.

The described problem can be extended easily in the way that it is not possible to achieve an optimal speed-up. We simply introduce the rule that each product has to be unregistered in a global registry, and that only one worker in each time step can unregister one product, see Fig. 1.2.

Now, the time for carrying one product to the van is t_1 and the time for the unregistering process is t_2. If we simply transfer the above algorithm then the van is completely loaded as soon as the last worker has loaded the n-th product. Hence, the total time is $n/p(pt_2 + t_1) = nt_2 + nt_1/p$, because for each of his n/p products the last worker has to wait $(p-1)t_2$ until he can

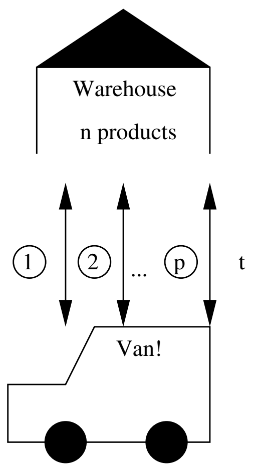


Figure 1.1: Simple warehouse example: Each worker has to carry n/p products from the warehouse to the van.

unregister his current product, which costs an additional time t_2. However, the optimal time would be $n/p(t_1 + t_2)$. Clearly, in this case, the registry is the bottleneck. Each worker has to unregister its actual product, carries it to the van and then returns to fetch the next product. If the time needed for the unregistering process is much bigger than the time to carry the product to the van and to fetch a new one (and there are many complex warehouse programs around) then the total time needed gets close to the time only one worker would need. As we can see in this worst-case scenario, the in-

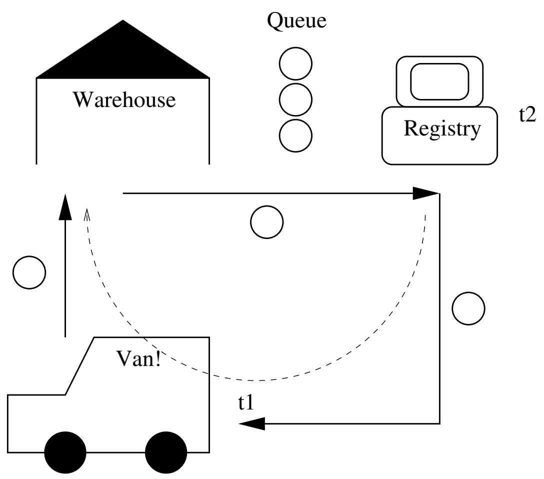


Figure 1.2: Simple warehouse example: Each worker has to carry n/p products from the warehouse to the van. Additionally, each product has to be unregistered.

vestment in p workers might be useless depending on the structure of the job. On the other hand, a smart warehouse manager would try to invest in his warehouse software to reduce the time t_2 or he would spend money in several registry workstations (e.g. p) to break the bottleneck and to increase the speed-up. Actually, the described process is nothing else than designing a *parallel algorithm*.

Now, the example does only cover the problem of distributing the work on several workers. After the distribution, the workers can do their job independently. Unfortunately, in the majority of problems it is necessary that the workers have to communicate during the execution of their job. They need partial results produced by the other workers, or knowledge only one worker has. Hence, the minimization of *communication cost* is a crucial issue for the design of a parallel algorithm. Assuming, that in our example there is one worker who has to tell all the other workers each time they come back to the warehouse which product they have to fetch next (perhaps because he is the only person who has the product list). We call this worker the boss. Let us denote the time needed for this communication as t_3. Then the total time for loading the van is $nt_1/p + nt_2 + nt_3/p$, because in each of the n/p steps such a communication operation has to be done. Therefore, it is obvious that t_3 has great influence on the total time. Now the questions are, which way of communication reduces t_3 maximally? Moreover, how much time does the communication take?

One possibility is that the boss tells each worker separately the next product he has to fetch. Let c be the time needed to tell a worker which product he has to take next. Hence, in this algorithm $t_3 = pc$, if we assume that he has to tell himself the next product, too (see Fig. 1.3).

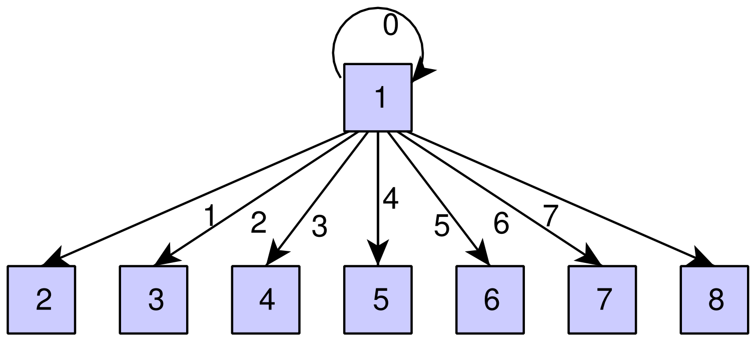


Figure 1.3: Simple Communication Strategy ($p = 8$): The boss (node 1) tells each worker separately which product he has to fetch next. The numbers at the directed edges denote the time step at which the call is done.

Another possibility is that the boss tells only one worker what they and the rest have to do. Then again each of the workers who know which products have to be fetched next (incl. the boss) chooses one additional worker and tells him what he and the rest have to do. This is repeated until all workers know which product to take next. The described communication pattern resembles a tree with p nodes and is illustrated in Fig.1.4. The height of the tree represents the number of time steps needed to perform the communication operation. Since in each step, the number of processors that know which products have to be fetched is doubled and each step takes time c, the total time for this operation is $c\lfloor \log p \rfloor$, which is better than the time for the last method.

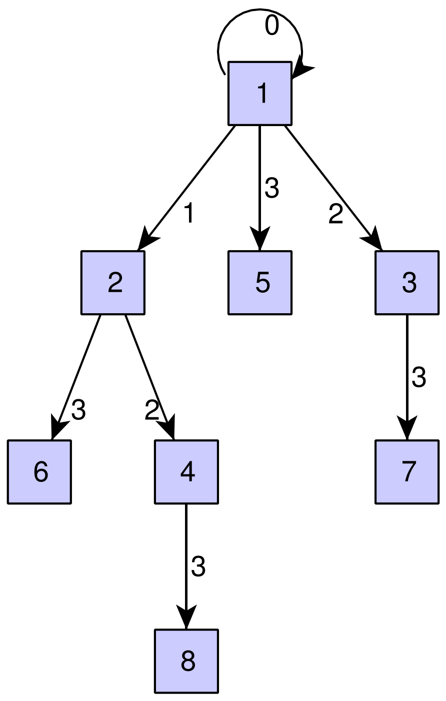


Figure 1.4: Tree Communication Strategy ($p = 8$): The boss (node 1) starts telling the next products to another worker. Then each worker (incl. the boss) tells another innocent worker which product to fetch. This is repeated until all workers know what to fetch next. The numbers at the directed edges denote the time step at which the call is done.

Although the example seemingly has nothing to do with computers, it introduces some main concepts of parallel computing as there is the problem of *load balancing*, the problem of achieving as much *speed-up* as possible, and last but not least, the problem of minimizing *communication cost*.

Looking at the area of computer science, there is no unique mapping for the workers to a certain object. For example concerning parallel algorithms, the workers would have been denoted as processors. However, a processor is just a piece of hardware which can be used to make calculations. Hence, it may be better to say that a worker is a process which works on a processor. Nevertheless, perhaps the processor itself is not just one piece of hardware it may consist of several smaller units which are able to perform micro-operations of the process in parallel. Therefore, within a certain part of each process, again we have workers. Further, on a higher level, workers can also be regarded as applications consisting of multiple processes and working on sets of processors or computers. Again, the workers appear on a different level. In the following, we show in which levels parallelism can be exploited by computer programs and outline which is the level of interest for the thesis.

1.2 Levels of Parallelism in Computer Programs

Concerning the execution of computer programs, the use of parallelism can be recognized in all levels, starting from applications down to microprocessor instructions [74].

1. Application level: Several applications can be executed in parallel on a set of computers. They all have their own working environment and do not know anything from each other. Normally, they even do not work on the same problem. Operating systems are responsible for the parallel or scheduled execution of such applications.

2. Process level: Applications may consist of several processes. Largely simplified a process can be seen as running program with a set of resources, which includes a private memory space. Parallelism in this level means that the processes try to solve the same problem together. In order to speed-up the application, they are executed on different processors at the same time. While their execution they are able to communicate with each other which is generally necessary to solve the common problem.

3. Thread level: Processes may have several threads of execution. A thread is an activity within a process and has its own control flow but shares resources and memory space with other threads in the same process. With the help of the operating system, it is possible to map the threads to different processors. Hence, several threads are executed in parallel and are able to communicate using the shared-memory space of the process.

4. Microprocessor instruction level: Microprocessor instructions can be executed in parallel, if the processor provides multiple pipelines for the execution of instructions (super-scalar). Optimizing compilers for super-scalar processors can analyze the instruction stream and are able to organize parallel executions by reordering of the instructions

in the streams. Another example for parallelism of instructions in this level are vector processors. They are able to perform the same operation on a vector of arguments at the same time. Consecutive vector operations can be executed in an overlapping parallel manner by the vector pipeline.

In the first level, operating systems are responsible for the execution and scheduling of applications. Parallelism is not used to reduce the time to solve a common problem; it is used to provide multitasking on computers which is a very important feature for workstations and servers.

The fourth level is a very low-level parallelism and is therefore normally done by compilers.

Hence, the levels of interests for developers of parallel programs are two and three, and this is the topic of this thesis. In these levels, it is possible to reduce the time for solving a problem by using several processes and threads on multiple processors or computers.

In the following sections, we will discuss the elements involved in the process of building a parallel application and we will show their interactions and interfaces.

1.3 Design Chain for Efficient Parallel Applications

The development of parallel algorithms and applications is based on several assumptions on the existing environment. The elements that play a key-role for the design of an efficient parallel application are illustrated in Fig. 1.5 and explained next.

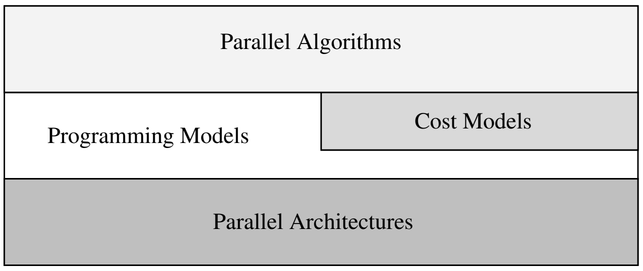


Figure 1.5: Elements of the parallel design process

- First, there is the *parallel computer* or *supercomputer* itself. There are many different architectures for parallel computers. Each *parallel architecture* has its own characteristics, which should be considered by a program. Critical issues are how the processors are connected to the memory banks, what kind of processors are used, or how fast they can communicate. We give a detailed overview in Chapter 2.

- Second, there are several *programming models* for the design of parallel programs. A programming model defines the way processes or threads can communicate or synchronize on the parallel computer. For each model, there might be several different libraries and each library has to be implemented for each parallel computer. Several programming languages can be used, but the focus in the area of high-performance computing is on C/C++ and FORTRAN. The closer a programming model fits to the underlying hardware the more efficient the implementation of the programming model can be realized.

- Third, a more abstract representation of the programming model is necessary with which algorithms can be designed and analyzed theoretically. We call these models *cost models*. They try to reflect the main characteristics of the underlying programming model and parallel architecture in a way that a significant cost analysis of an algorithm is possible. Hence, algorithms are getting comparable and running times are getting predictable only by the results of their theoretical analysis.

- The fourth key-role element are parallel algorithms. They are the building blocks of parallel libraries and applications. It is very important to develop efficient algorithms for problems, which appear very often, in order to offer a large collection for the development of complex applications.

Ideally, algorithms are designed on top of cost models. These cost models reflect the main characteristics of the underlying parallel architecture and therefore the analysis is a good estimate for the running time of following implementations. In order to implement algorithms developed on cost models, appropriate programming models must exist that implement the cost models. The programming models themselves consist of programming languages and libraries, and should fit to the underlying architecture as much as possible. In practice, programmers very often develop and implement algorithms using programming models directly. In this case, the usage of a cost model is implicitly done through the programming model. Hence, there is no accurate analysis of the algorithm, which implies the risk of being inefficient.

Obviously, in this chain from the algorithm to the architecture there might be loss of performance if the various levels do not fit properly. For example if the cost model is inadequate towards the architecture because it assumes an infinite memory space or an infinite number of processors, then the resulting algorithms might be useless or at least inefficient in practice for the underlying architecture. There might also be problems if the programming model does not fit to the architecture. If it assumes a shared-memory for all processors but the underlying architecture consists of distributed memories, then there will be a loss of performance because the implementation of the programming model has to simulate a shared-memory.

Hence, there is a gap between parallel algorithms and real parallel machines and the models should try to build a bridge. A lot of important work has been done in order to build general-purpose cost models to overcome this gap. These models were called parallel bridging models. We will describe the most popular in Chapter 2.3.

Nearly independent of this process a huge variety of programming models were developed. The most important will be reviewed in Chapter 2.4.

1.4 The Contribution of the Thesis

In the center of the thesis, there is a parallel architecture called Clusters of Symmetric Multi-Processors (*SMP Clusters*). This is a hybrid architecture concerning the way processors are connected to each other. The building blocks are multiprocessor nodes with shared-memory. These nodes are connected by an interconnection network. Hence, there is parallelism within the nodes exploiting shared-memory and between the nodes using the interconnection network, which creates a *parallel hierarchy* (see Fig. 1.6). Basically, the communication between nodes is more expensive than within the nodes which leads to a *hierarchical communication architecture*.

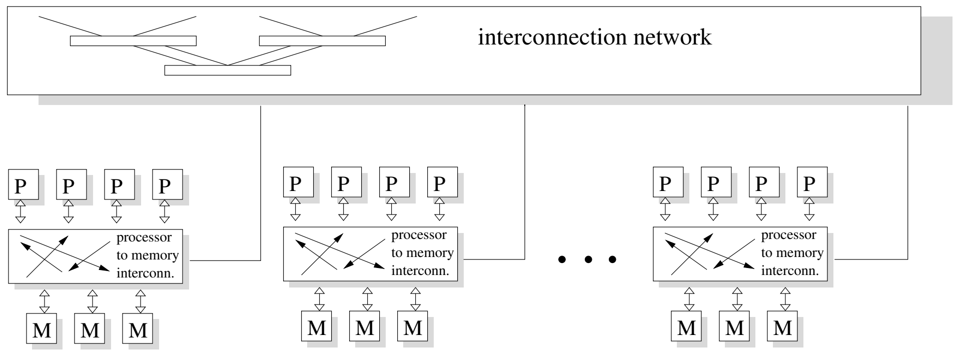


Figure 1.6: Outline of the SMP Cluster architecture

There has been an enormous trend towards this architecture in the last years because of economical and performance reasons. Together with the emerging of this architecture a debate has been started which programming model is the best for it. There are several approaches, which will be reviewed in Chapter 2.4.2. Most of these approaches just ignore the new communication hierarchy and make it transparent to the algorithm and application developer. One advantage of such a strategy is that existing programs can easily be ported by more or less recompiling. Another advantage is that a huge variety of programming models and libraries are thus available for the architecture.

On the other hand stopping the consideration of the architecture at the programming model level means to waste the chance of optimizing on the algorithmic level and thus, to get the maximal performance possible. Hence, the main part of the thesis it to show possibilities for algorithmic optimization on SMP Clusters. As explained above, the main difference of SMP Clusters is based in the communication hierarchy and the parallel hierarchy. Therefore, all optimizations presented in the thesis are based on communication operations and different data distributions.

According to the design chain for parallel applications depicted in Section 1.3, we define a cost model for SMP Clusters. This cost model reflects the main characteristics of SMP Clusters and of the *hybrid-programming model* that will be presented in Chapter 2.4.2. This model suggests using one process per SMP node. The processes can communicate over the interconnection network. Within the processes, multiple threads are responsible for the parallel execution. As mentioned earlier, there has been a lot of work in finding a general-purpose cost model for all architectures. Indeed, these models reflect the most important characteristics of parallel computers but not the communication hierarchy and the hybrid memory model of SMP Clusters. Despite of that our approach is to extend the most accepted general-purpose cost model named bulk synchronous parallel (BSP) model [76] with the characteristics of SMP Clusters.

The described design chain provides a quantitative basis for the analysis of algorithms and guarantees by the similarity of cost model, programming model and architecture that a transfer to an efficient implementation is possible. Although this is a pre-condition for developing parallel applications, it is not a methodology how parallel algorithms can be designed. Hence, there is a need for design methods that help the developer to create parallel algorithms for given problems. With the help of the methods parallel algorithms can be developed. By using the cost model for the analysis, different approaches can be evaluated and compared. In [31], a general methodology for developing parallel algorithms is presented. We use this methodology as a framework for the over-all design process of parallel algorithms for SMP clusters. We will review this methodology, show its usage for SMP clusters and present additional methods

- for the design of parallel algorithms from scratch,
- for an efficient transfer of parallel algorithms to SMP clusters and
- for optimizing distinct aspects of these algorithms.

All these methods can be regarded as additional modules of the over-all design methodology. They can be applied at different stages of the whole design process from the problem to a highly optimized algorithm for SMP clusters. Some only assume the problem definition (or a sequential algorithm) as an input, some a parallel algorithm and others can be applied to optimize SMP cluster algorithms. Hence, in order to design an optimized parallel algorithm for SMP clusters, the methods can be applied consecutively one after the other, which corresponds to a path through the set of methods.

We will explain generally all methods whereby the explanation is supported by simple examples. Further, we present case studies for more complex algorithms that present several paths through the design methods.

In each case study, we present theoretical, and if necessary, practical evaluations of the algorithms. The cost model serves as a super-model for the analysis. It is not always necessary to use the whole model for the analysis of the sample algorithms. For each case study, another reduced instance of the cost model is sufficient, which makes the analysis more feasible.

In this sense, the thesis is an attempt to show the possibility of unifying the theoretical and the practical aspects of developing parallel applications considering SMP Clusters as example.

1.5 Structure of the Thesis

- In Chapter 2, we will give an overview on parallel architectures, motivations for building SMP-Clusters, parallel bridging models and programming models for SMP-Clusters.
- We proceed in Chapter 3 with a theoretical cost model for SMP Clusters called κNUMA and show an analysis for broadcast problems.
- In Chapter 4, we review and present several methods for designing and optimizing parallel algorithms for SMP clusters.
- Chapter 5 is the first case study for designing algorithms for SMP Clusters. After using a method for transferring well-proven parallel algorithms, it is observed how data redundancies can be exploited in order to improve the performance of the algorithm. Hence, the case study is called *exploitation of redundant data*. The sample problem is the parallel dense matrix-vector-multiplication.
- Chapter 6 illustrates the second case study called *adaptation of communication patterns* using the example of parallel matrix transpose.
- In Chapter 7, the case study presents the method of *hierarchical-sensitive design*. We address to the problem of integer sorting using parallel radix sort.
- General conclusions can be found in Chapter 8.

Chapter 2

Overview on Parallel Architectures, Models and Libraries

2.1 Architectures

Computer architectures are classified according to their ability to realize the parallel program execution. Flynn's taxonomy [29] divides computer architectures into four categories depending on how the instruction streams and the data streams are implemented. A more actual interpretation of this taxonomy is given in [77].

1. Single instruction stream and single data stream (SISD): These are systems containing one CPU and therefore are able to execute one instruction stream serially. Large mainframes that consist of multiple processors do also belong to this class, because each processor executes unrelated instruction streams. Hence, they can be regarded as a couple of SISD machines working on different data spaces.

2. Single instruction stream and multiple data stream (SIMD): Systems of this class are characterized by a huge number of processors, ranging from thousands to ten-thousands. There is a global clock and in each lock-step all processors execute the same instruction on different data. In one lock-step, one instruction works on many data items in parallel.

3. Multiple instruction stream and single data stream (MISD): Theoretically, in such a machine different instructions are executed on the same data at the same time. Until now, no practical machine of this type has been constructed.

4. Multiple instruction stream and multiple data stream (MIMD): Several instructions are executed in parallel on different data. In contrast to the multi-processor SISD machines, the instruction streams and the data streams are related. They are all parts of a global program and hence work on the solution for a common problem. MIMD systems are used for running sub-tasks in parallel with the aim to reduce the time to solve the global problem. Most systems belong to this class and hence a further criteria will be introduced later to make a more accurate classification.

Although Flynn's taxonomy is very useful, it does not cover how processors are connected to each other and to memory units. However, parallel architectures are made up from multiple processors and memory units.

Networks that are connecting processors with memory units are called *dynamic networks* [52] , because they are built using switches and communication links. Paths among processors and memory banks are established by connecting communication links dynamically using the switching elements. Extreme but realistic examples of these networks are the $n \times n$ *crossbar*, see Fig. 2.1 , and the *bus connection*, see Fig. 2.2. The former is a common hardware channel that can link only a pair of modules at a time. It has the least circuital complexity and cost. The latter is a square matrix of switches that can connect up to n non-conflicting pairs of modules. It achieves the highest connectivity at the highest circuital cost.

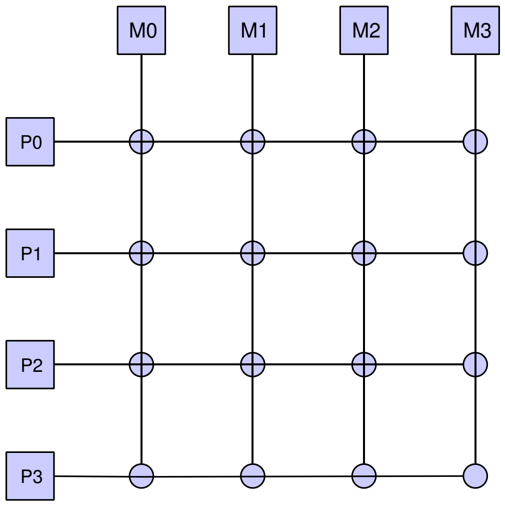


Figure 2.1: $n \times n$ crossbar switch, with $n = 4$.

Many systems use structures that are in between these two, as a compromise between practical scalability and performance. These networks are called *multi-stage crossbars*. They are networks made of smaller interconnected crossbars. The *omega network* is illustrated in Fig. 2.3 as an exam-

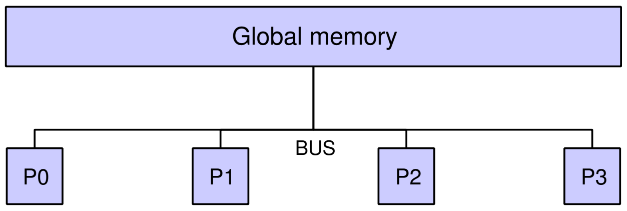


Figure 2.2: Bus-based architecture with no cache

ple for a multi-stage network. The advantage of such networks in contrast to complete crossbar is that it only needs $(n/2)\log n$ switching elements instead of n^2.

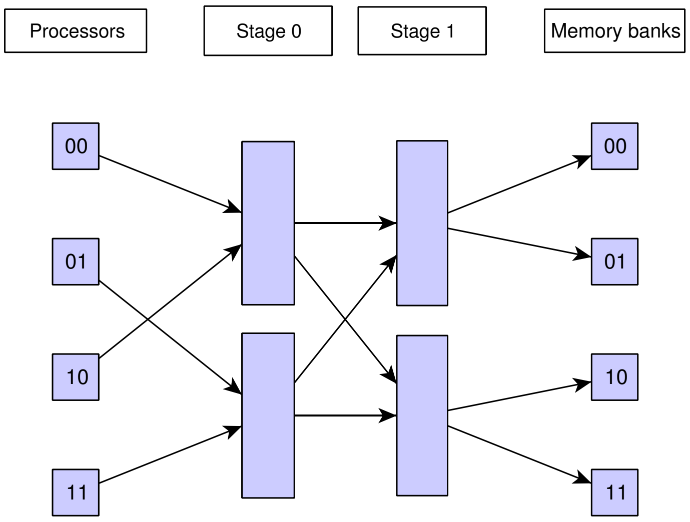


Figure 2.3: Example for a multi-stage crossbar: A complete omega network connecting four processors with four memory banks.

Networks which are connecting processors are called *static networks*, because they consist of point-to-point communication links among processors. Typical structures for these networks among others are the Fat-tree, Hypercube, 2D or 3D Meshes, and multi-stage networks like Butterfly, Omega, or Clos networks. Fig. 2.4 shows eight processors connected by a 4D hypercube interconnection network. These and derived networks are in use within a wide range of parallel computers and network elements like e.g. hi-speed-switches. Detailed descriptions and design issues of static and dynamic networks are discussed in [52, 53].

Efficiency of communication is measured by two parameters, *latency* and *bandwidth*. Latency is the time taken for a communication to complete, and bandwidth is the rate at which data can be communicated. Latency can also be imagined as the time needed to communicate zero data. In a

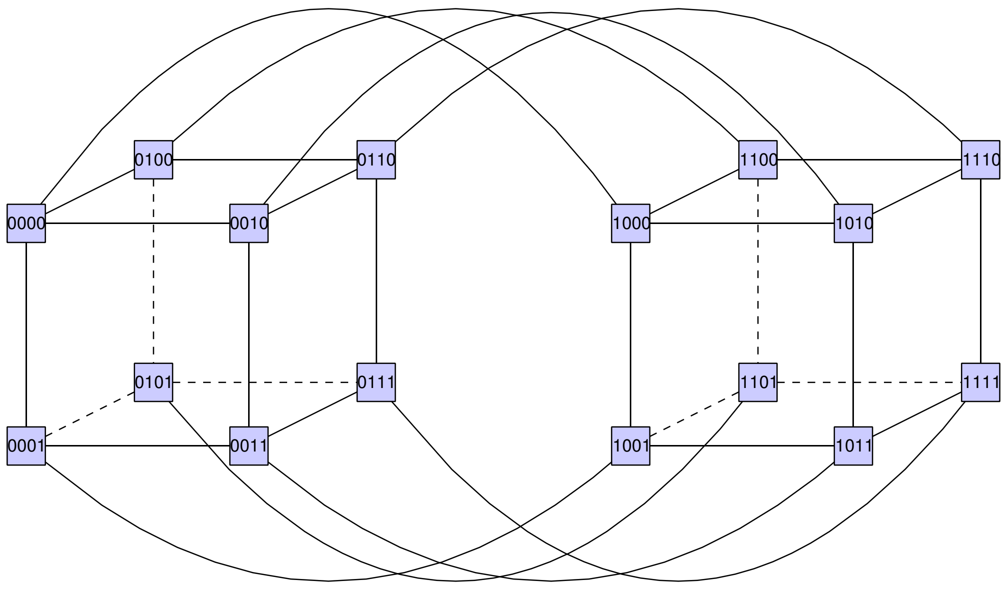


Figure 2.4: Example for a static network: A 4D hypercube connected architecture.

simple world, these metrics are directly related. For communication over a network, however, we must take into account several factors like physical limitations, communication start-up and clean-up times, and the possible performance penalty from many simultaneous communications through the network. A rule of thumb is that latency depends on the network geometry and implementation, and bandwidth increases with the length of the message, because of the decreasing influence of fixed overheads.

As already mentioned above, almost all modern parallel computers belong to the MIMD class of parallel architectures. Basically, this means that processing nodes can execute independent programs over possibly different data. The MIMD class is subdivided in [77] 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, which is actively supported by the network hardware (see Fig. 2.5a). Different processors can interfere with each other when accessing the same memory module, and race conditions may show up in the behavior of the programs. Therefore, hardware lock and update protocols have to be used to avoid inconsistencies in memory and among the caches [1]. Choosing the right network structure and protocols are critical design issues, which drive the performance of the memory system. Larger and larger shared-memory

[1]We do neither analyze in depth here cache coherence issues, nor multi-stage networks, nor cache-only architectures [52].

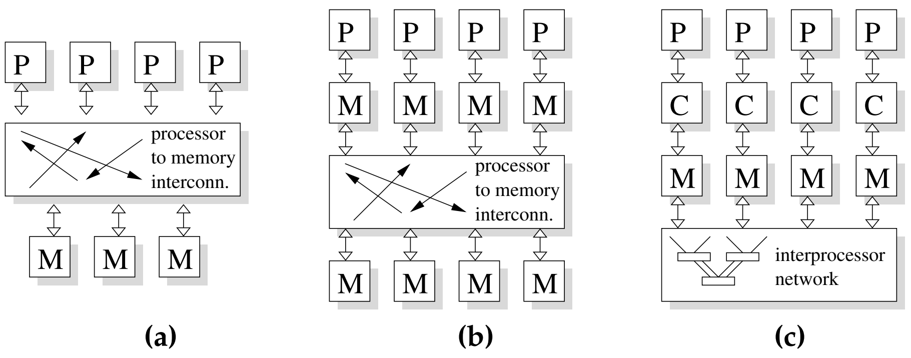


Figure 2.5: Overall structure of DM and SM MIMD architectures. (a) Generic SM MIMD machines have multiple processors and memory banks (not necessarily the same number) – (b) example of NUMA SM MIMD with multiple local and global memory banks, and a bus interconnection – (c) example of a MIMD architecture composed of single-processor nodes and a more sophisticated interconnection network. Depending on the network implementation, this can be either a DM-MIMD or a NUMA SM-MIMD architecture.

machines lead to difficult performance problems.

Multi-stage crossbars are getting more and more important with an increasing number of processors in shared-memory machines. If a multi-stage crossbar connects the processors and each processor has some local memory banks, there is a memory hierarchy within the shared-memory of the system (see Fig. 2.5b).

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

In contrast to the shared-memory machines, in a distributed-memory machine each processing node has its own address space. Therefore, it is up to the user to define efficient data decompositions and explicit data exchange patterns for the applications. Each processing node has its own local memory, so distributed memory architectures obviously belong to the NUMA architectural class (see Fig. 2.5c). The network is inherently slower than local memory; hence, we have a memory hierarchy in DM MIMD machines, too. However, distributed-memory architectures are less demanding with respect to the interconnection network [2] , so they are much more

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

Classification	count	share
MPP	211	42.2 %
Cluster	149	29.8 %
Constellations	139	27.8 %
SMP	1	0.2 %

Table 2.1: Summary of the TOP500 list (June 2003) of the fastest supercomputers according to the architectural classification.

scalable than the shared-memory ones.

In recent years, a strong trend has emerged in the field of high performance computers towards two kinds of architectures, (1) clusters of vector computers and (2) clusters of scalar uni- and multiprocessors. Looking at the list of the fastest 500 supercomputers in the world [60], the majority of them belongs to these two classes, with the latter steadily gaining more share.

Especially clusters of SMP nodes (*SMP clusters*) are a more and more emerging architecture for building parallel computers. In the latest list of the fastest 500 supercomputers (June 2003), 29.8 % of the supercomputers were classified as clusters, whereby de facto all these systems are clusters of SMP nodes. Additionally, SMP clusters are also found in the other classes of the summary of the TOP500 list depicted in Table 2.1. Bell and Gray [8] define *constellations* as clusters of nodes with larger shared-memory multiprocessor nodes, where each node is more powerful than the casual PC uni- or dual-processor nodes. Additionally, clusters of vector processors do also belong to this class, because the performance of modern vector processors is comparable to that of larger SMP nodes. *Massively parallel processing systems (MPP)* can be characterized roughly by a very large number of processors with local memory that are connected by a special interconnection network. Nevertheless, despite of that, systems that clearly have the SMP cluster structure were classified as MPP systems in the list. For example, the number four in the current list, the *IBM ASCI White*, is an IBM RS/6000 SP system that consists of 512 nodes and each node has 16 Power3 processors. Probably this system was classified as a MPP system, because the number of 8192 processors is very high and typical for MPP systems. Hence, much more than 30% of the supercomputers in the list are SMP clusters. Further, 8 out of the fastest 10 supercomputers are SMP clusters, too. We did not count the number one of the list, the Earth-Simulator, because its 640 8-way nodes consist of vector processors. However, despite of that, these numbers clearly underline the importance of the SMP cluster architecture.

At different scales, these SMP cluster systems can be classified both as DM and as 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, we could classify SMP clusters either as SM or as DM MIMDs depending on the existence of a common address space abstraction for all the processors, eventually provided by firmware or software layers. However, even if a shared space is provided this way, algorithms that exploit memory locality within SMP nodes incur much fewer communication overheads, and can achieve a better 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 meta-computing technologies, where supercomputers or clusters of workstations are connected with each other to run applications, result in even more levels and a less regular parallel hierarchy. Broadband connections, ranging from local area networks to geographic ones, add more levels to the hierarchy, with different communication bandwidth and latency [32].

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

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

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

The effects on latency and bandwidth of the parallel hierarchy are similar and combine with those of the ordinary memory hierarchy, consisting of several levels starting at the processor's registers up to the hard disks. A crucial observation is that there is no strict order among the levels of these two hierarchies, which we can easily exploit to build a unitary model. For instance in some systems, we can see that the communication layers (both SM and DM based ones) provide a bandwidth lower than main memory, and in some cases lower than that of local I/O. However, their latency is usually much lower than that of mechanical devices like disks. Thus, different access patterns lead to different relative performances of communication and I/O.

Assessing the present and future characteristics of the parallel hierarchy [19] and devising appropriate cost and programming models to exploit it are among the main open issues in modern parallel/distributed computing research.

2.1.1 Motivation and Technological Perspective

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

In [48] some main advantages of SMP cluster architectures are found, most of them being technological and economical considerations.

- Standard off-the-shelf processors are getting faster and faster, even with respect to special purpose architectures. Because of 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 massively parallel processing machines (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 MPPs. 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, and it is practically expandable by adding more nodes and/or upgrading processing nodes. While it is usually 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 critical resources in a supercomputer. A greater total memory size and number of disks characterize SMP clusters. Hence, it is possible to have more active jobs, which even have larger data storage available.

- Most software for MPPs or SMPs can easily be ported to SMP clusters achieving similar efficiency. With the knowledge of software for SMP machines and the already existing software for MPPs, it should be possible to provide a powerful environment for parallel software development and execution. We will give an overview of the efforts in this direction in Section 2.4.

Some of the preceding considerations have been recognized years ago, while others are a more recent discovery. According to Bell and Gray [8], the trend will last for more than a while. They depict a scenario of the evolution of parallel computer and computing grid architectures, which is described below.

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 that 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 [32] or meta-computing 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 are 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. In this sense, this thesis can be regarded as one step in this effort.

2.2 Computational Cost Models

In sequential computing algorithms were designed using the classical random access machine (RAM) model which is based on the Von Neumann computer. Due to the more complex structure of modern computers, concerning memory hierarchy this model does not properly account with the cost of memory access. Hence, more complex multi-level computational models have been suggested to obtain better predictions of the algorithm's practical behavior. The most known of them is the PDM model [78].

In parallel computing, we have a similar situation. A first approach of modeling a parallel computer was to extend the RAM model with multiple processors using a shared-memory. The resulting computational cost model is called the parallel random access machine (PRAM) model [30]. It is based on several precise assumptions on the parallel computer:

- There is an unlimited number of processors (simple processing units with local memories) that run the same program. They are connected by an unlimited global shared-memory where they can read and write in parallel.
- PRAM machines can be differentiated according to their capability of accessing the shared-memory. Read and write operations can be either exclusive or concurrent. In case of a concurrent write rules are defined what happens to the memory location if several processors manipulate it in the same time step.
- The execution of a program is done synchronously. There is a global clock and in each time step, all active processors always complete one instruction.

Hence, the PRAM model assumes the same costs for a computation step, a local memory access and a global memory access. Practical aspects like the memory hierarchy and bandwidth constraints due to certain interconnection networks are completely ignored. The focus is solely set on concurrent program execution. These assumptions cause the model to be very independent of the properties of a specific architecture, and hence the PRAM model can be used as an effective model for analyzing the abstract computational complexity of problems. On the other hand, these assumptions are not realistic for the majority of architectures described in Section 2.1. Real MIMD machines are much more complex and hence using the PRAM model contains the risk of getting misguiding results with respect to real computational costs.

Several extensions have been developed for the PRAM model in order to get the theoretical computational cost closer to real performance. A survey on these derived models is given in [35].

A second research part in parallelism considers efficiency of communication in interconnection networks. Different network structures are compared according to their ability to perform communication operations and costs concerning hardware elements. It is possible to embed certain network structures into others and hence to simulate efficient networks algorithms in the environment of other structures. Despite of that, 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.

A third research track started in the 1990s introducing the class of parallel bridging computational cost models. These models try to describe the main properties of complex parallel architectures without being too complex or being too imprecise. On the one hand, these models do not want to be architecture-specific, but on the other side they want to predict the real performance of a program as accurate as possible. These models are subject of the following section.

2.3 Parallel Bridging Models

Computational cost models have to find the right balance between abstraction and accuracy. They should reflect the real behavior of parallel algorithms without making the analysis too complex. A parallel cost model can be regarded as useful, if the results of analyses can be confirmed in practice. A model that achieves this aim without being limited to a special architecture is called *parallel bridging model*. A PBM separates the development of efficient algorithms and software from the underlying architecture. Several models of this class have been developed in the 1990s. They use a more abstract approach in modeling the interconnection architecture.

The most well-known and accepted PBM was introduced by Valiant [76]. The following goals of a PBM were formulated in the context of its initial definition.

Cost measure A PBM has to define a cost measure that describes the individual operation costs and hence, guides the development of algorithms. In fact, the cost measure is the main element of the model because it defines which characteristic of a parallel computer is considered. The model should be independent of a specific architecture and technology, but it should reflect the fundamental constraints of parallel machines.

Efficient universality Implementations of PBM algorithms on real machines should not lead to great loss of performance. While logarithmic simulation losses have to be avoided, constant bounded inefficiency is within the tolerance.

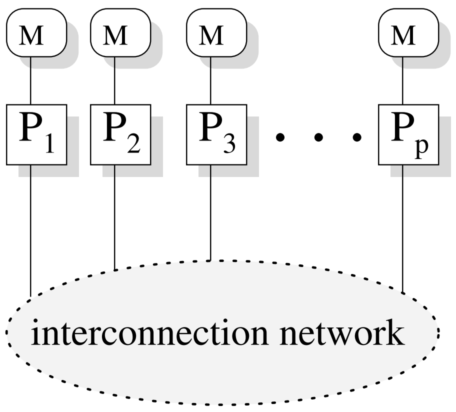


Figure 2.6: The BSP abstract architecture.

Neutrality A PBM has to be neutral concerning the number of processors. Although the results are expressed asymptotically, they should be applicable to ranges from only a few to millions of processors. Hence, an approximate result is justified if the factors are small.

Portability The programmer should not have to care about low-level problems like explicit memory management, complex communication operation or synchronizations. Thus, there has to be a programming model, which is close to the PBM supporting high-level functions.

Parallel slackness A PBM algorithm designed for v virtual processors should be optimally simulated on p physical processors if p is smaller than v (e.g. $v = p \log p$). This enables to overlap communication and computation of different virtual processors on a wide range of interconnection networks.

2.3.1 The Bulk Synchronous Parallel Model

The *bulk synchronous parallel model* (BSP) (as described in [3]) is a set of processors with local memories and a complete interconnection network, see (Fig. 2.6). A router delivers messages between pairs of processors. The model uses three parameters to describe a BSP computer, see Fig. 2.7. The number p of processors, a latency parameter L, which is the maximum latency of a message or synchronization in the network, and a parameter g, which is the basic throughput of the network or the bandwidth inefficiency.

A BSP computation consists of *supersteps*. During a superstep, processors can do computations on their data in local memory and can send and receive a certain amount of messages with each other. Messages sent during superstep t are received only at the beginning of superstep $t+1$. Fig. 2.8 shows the phases of a superstep. Each superstep consists of computation

p	number of processors
L	message latency / synchronization
g	cost parameter for message-passing
for processor i in superstep t	
$w_{i,t}$	local computation
$\lambda_{i,t}$	num. of sent messages
$\mu_{i,t}$	num. of 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

Figure 2.7: BSP symbols and parameters.

phase (gray stripes), a communication phase with varying communication pattern, and the constant bounded synchronization time L (white stripes).

The table of Fig. 2.7 summarizes the composition of the costs in the BSP model for a superstep t and a processing node i.

Let $h_{i,t} = \max(\lambda_{i,t}, \mu_{i,t})$ be the largest number of messages sent or received by processor i during the current superstep t. The communication pattern (set of all point-to-point communications in a superstep) has size $h_t = \max_i h_{i,t}$ for superstep t, and is called *h-relation*.

With these parameter definitions, w_t and h_t are the largest w and h values in superstep t.

Hence, the total time of a superstep is $w_t + g \cdot h_t + L$, because , this can be regarded as an upper bound, if we assume that synchronization after each superstep is only enforced when actually needed for the correctness of the algorithm. 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*

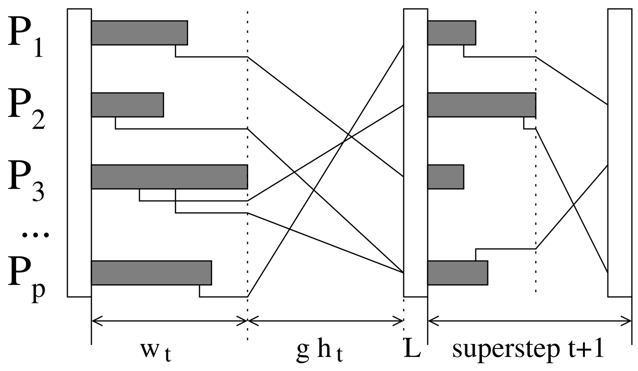


Figure 2.8: BSP superstep execution

$W = \sum_t w_t$ and the *communication volume* $H = \sum_t h_t$ of the algorithm lead to the cost estimate $W + g \cdot H + L \cdot T$.

Another useful criterion for judging BSP algorithms is the *c-optimality*. If T_{seq} is the time of the best known sequential algorithm then a BSP algorithm is called *c-optimal*, if $W = c \cdot T_{seq}/p$ and $g \cdot H + L \cdot T = o(T_{seq}/p)$ for a small constant $c \geq 1$.

In the next section, we will address to other parallel bridging models. We will present the CGM model, which is the closest to the BSP, the LogP and the QSM model.

The Coarse-Grained Multicomputer

The *coarse-grained multicomputer* (CGM) model is similar to the BSP model concerning the superstep-based execution scheme of a parallel algorithm. In contrast to BSP, there are only two parameters. The number of processors p, and the problem size, n. Each node has $O(n/p)$ local memory. No assumptions are made for the network structure. Algorithms exploit a reduced number of fixed, parallel primitives for computation and communication, which can be efficiently implemented over various interconnection networks.

Each CGM algorithm has a parametric cost depending on n and p. Further, the costs for the used primitives depending on the problem size n are added. In Fig. 2.9, they are represented by f, g, s. Depending on the network topology the time for each primitive can be inserted (e.g. complexity of exchanging $O(n/p)$ keys in a hypercube of diameter $\log_2 p$, see Fig. 2.4 for an example of an hypercubic structure).

A CGM superstep consists of three phases. The decomposition phase (the work is distributed using parallel primitives), the local computation phase, and a merge phase (results of local computation are communicated) also exploiting the parallel primitives. The task of formulating a CGM al-

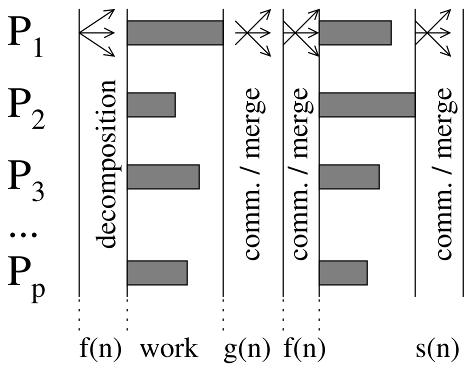


Figure 2.9: CGM supersteps

gorithm is to decompose the problem into coarse-grain independent subproblems using these global portable parallel primitives. The algorithm, which needs the smallest number of supersteps is the best.

During the years, in the common use CGM has become close to BSP. In recent works (e.g. [25, 26]), CGM algorithms are often defined as a special class of BSP algorithms.

2.3.2 LogP Model

The LogP model [24] ignores the network geometry like the BSP model. The processors communicate through point-to-point messages. The model defines four parameters. An upper bound on communication latency L, the overhead involved in a communication (for sending and receiving messages) o, a gap parameter g (the time a processor has to wait before the next communication can be started), and the number of processors P. As we can see the notion of the parameters are responsible for the name LogP. In the original model (*stalling* LogP) an explicit capacity constraint is assumed for the network. No processor can have more than $\lceil l/g \rceil$ messages in transit to it at the same time. Senders that hurt the constraint are assumed to stall. *Non-stalling* LogP dismisses the network capacity constraint. Therefore, in the stalling model the messages are considered to be of small fixed length.

The design and analysis of LogP algorithm is more complex than for other models, because of the unstructured and asynchronous nature and the capacity constraint requirement. There are comparably fewer results with LogP, even if most basic algorithms (e.g. broadcasts or summing) have been analyzed in great detail. An outline of the LogP-broadcast is presented in Fig. 2.10.

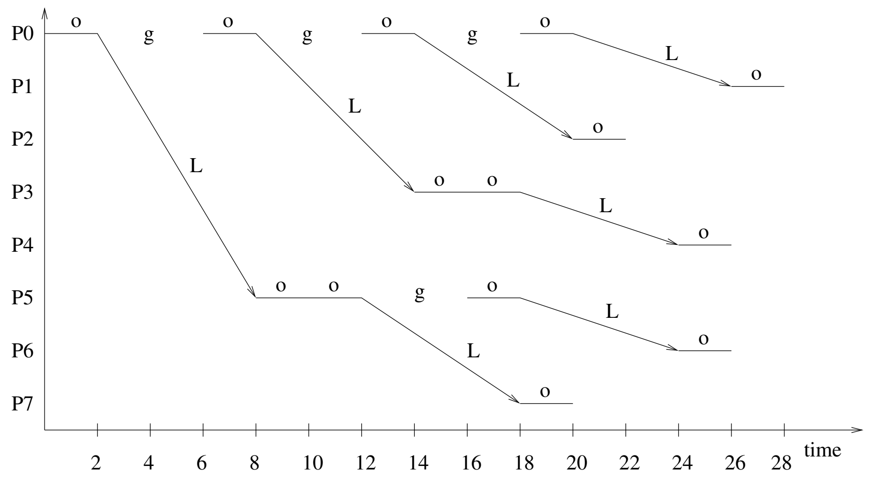


Figure 2.10: LogP-Model: Optimal broadcast tree for $P = 8$, $o = 2$, $g = 4$ and $L = 6$. Processor P0 is the initiator of the broadcast.

2.3.3 QSM model

The *queuing shared-memory* (QSM) [34] model can be seen both as a PRAM evolution and as a shared-memory variant of the BSP. Each processor has a local memory. The processors communicate by reading and writing to a shared global memory. The execution of an algorithm is divided into phases, with reads and writes. The reads and writes to the shared-memory are posted at the end of each phase. Concurrent read (or writes, but not both) to a memory location are allowed.

Each processor performs 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 the 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 same meaning as in the LogP model , 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.

2.3.4 A Comparison of Parallel Bridging Models

It is possible to compare the models by emulating one model on the other. Emulations are *work-preserving* if the product $p \cdot t$ (processors per execution time) on the emulating machine is the same (within a constant factor) as that on the machine being emulated. If the emulating machine has fewer processors than the emulated one, the emulation is characterized by a certain slowdown. This slowdown is $O(f)$, if we are able to map an algorithm, running in t time on p processors, to one running on $p' \leq p/f$ processors in time $t' = O(t \cdot (p/p'))$. If the slowdown is $O(1)$ then the emulation has at most a constant factor of inefficiency.

Ramachandran [62] presents asymptotic slowdown results in a recent survey. If there are several work-preserving emulations with small slowdown, this indicates that these models are equivalent in their use as cost models for real parallel machines.

But despite of that, some models are preferable since there exist adequate programming models for them and since a more abstract view of the algorithm's structure and communication patterns allows an easier design and analysis.

From this point of view, the LogP and the QSM are less suitable, because they provide too much low-level analysis concerning communication behavior and memory access. It is very difficult to analyze complex algorithms with these models. The QSM can be used to evaluate the practical performance of many existing PRAM algorithms. A disadvantage of QSM is that it disregards hierarchical structures of the computation, and it has an abstract but fine-grain approach to communication cost.

In contrast, the BSP and the CGM can be used easier for the design of parallel algorithms, because of their coarse grain nature and their abstraction of complex communication patterns. Further, several programming models and software tools have been designed with which it is possible to implement BSP algorithms directly. The two main libraries are the Paderborn University BSP library (PUB) [15] and the Oxford BSPlib [42]. These are special libraries, which pretend to close the gap between the BSP model and real parallel machines. Unfortunately, these programming models do not play a major role in practical parallel programming. However, BSP algorithms can also be implemented using more "standard" programming models, which we will describe in section 2.4.

In the following section, we will have a look on extensions of the BSP model with respect to hierarchical network structures.

2.3.5 Parallel Bridging Models and Hierarchical Parallelism

After the introduction of PBMs, research in this area focused on enhencing their accuracy. This is done by exploiting the concepts of processor locality and block-oriented communications.

In Section 2.1, we showed that parallel architectures consist of regular hierarchical structures. These architectures reward accesses to local data. The closer the data is to the processor, the faster is the access. Parallel Bridging models can be extended by introducing parameters that indirectly reflect actual communication behavior without making explicit assumptions on the underlying architecture like e.g. geometry.

In the following two extensions of the BSP model are presented. Its abstract and simple communication model is extended first by caring about effects on the message length and second by considering the relationship between network size and parallel overhead in communications.

The BSP* model.

In contrast to the BSP communication model, communication in real interconnection networks is dependent of the message length. BSP does not account for this characteristic of communication. BSP roughly considers congestion effects in networks by involving the number of exchanged messages. The combination of bandwidth constraints, start-up costs and latency effects is often modeled as a linear affine function of message length. To model the practical constraint of efficiency for real communications, the BSP* model [3] 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. This parameter is integrated in the cost function of communication. The function has to account for both, the number of messages in a superstep h_t, and the communication volume s_t (the sum of all sizes of all messages). Each message is charged a constant overhead, and a time proportional to its length in blocks. The costs of a superstep are defined as $w_t + g(s_t/b + h_t) + L$, often written as $w_t + g^* \cdot (s_t + h_t \cdot b) + L$, where $g^* = g/b$. This has the effect that algorithms that produce high communication volumes and long messages are not rewarded. Nevertheless, due to the parameter b, too many small messages do also have a negative impact on performance. Both aspects are close to real communication behavior.

Summarized, BSP* encourages block-organized communication and a reduced amount of data transfers.

D-BSP model.

The behavior of the BSP model is independent of the size of the BSP machine. Thus, the model's behavior is adapted if parameter values are changing (e.g. adding processors to a bus interconnection leads to larger values of g and L). On the other hand, there is no way we can model situations that are more complex 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, in comparison 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.

The *decomposable* BSP model (D-BSP) was introduced by De La Torre and Kruskal [73]. D-BSP rewards locality of computation by allowing hierarchical decomposition of the machine into smaller BSP-sub-machines.The g and L parameters of BSP are replaced with two functions, $\mathcal{G}_m$ and $\mathcal{L}_m$, of the sub-machine size m.

During algorithm execution, the computation can be recursively split, and smaller sub-problems can be assigned to different sub-machines. The sub-computations are still D-BSP computations, which proceed independently until they merge again. Their computational cost is the maximum of the costs of the sub-computations, and each is evaluated with the 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.

Measuring communication cost in this abstract way has the advantage that D-BSP does not have to deal with the geometry of the interconnection structure directly. It only deals with characteristic functions that are representing the interconnection structures. Hence, D-BSP adds the ability to consider architectural effects on communication due to the network.

A first comparison among the D-BSP and BSP models can be found in [11]. Meyer auf der Heide and Wanka [3] investigated the relationships among the BSP* and the D-BSP models. Bilardi and others [12] also examine the D-BSP model, concluding that it offers the same design advantages of BSP, but has higher effectiveness and portability over realistic parallel architectures. They show results for the family of functions $\mathcal{G}_m, \mathcal{L}_m$ of the form $C \cdot (n/2^i)^\alpha$, where $m = 2^i, \quad (0 \leq i \leq \log n)$ and $(0 < \alpha < 1)$. These functions capture a wide family of commonly used interconnection networks with n nodes, including multidimensional arrays.

2.4 Programming Models

In this section, we will survey a set of programming models and software tools that can be used to exploit parallel hierarchical architectures. We will start with libraries used for parallel programming, and we will show improvement of these libraries that try to exploit the hierarchical structure of SMP Clusters. As we will see in the various approaches, writing programs that fully exploit hierarchical parallel architectures is a difficult and error-prone task, where a large part of the effort is spent in tuning and debugging activities.

2.4.1 Parallel Programming Libraries

There are two main parallel programming paradigms, which fit the two extremes of the MIMD architectural class, the message-passing paradigm for distributed-memory machines and the shared-memory paradigm for the shared-memory machines.

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

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

For both paradigms, there is one de facto standard library, respectively the *message-passing interface* (MPI) standard, and the OpenMP programming model for shared-memory programming.

Message-Passing-Interface MPI

In 1994, the MPI-Forum unified the most important concepts of message-passing-based programming interfaces into the MPI standard [56]. The current, upward compatible version of the standard is known as MPI-2 [58], and it specifies primitive bindings for languages of the C and FORTRAN families.

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

There are MPI implementations for nearly all platforms, which is the prerequisite for program portability. Key features of the MPI standard include the following:

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

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

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

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

  Well-known examples of collective operations are the *barrier synchronization* (processes wait for each other at a synchronization point), the *broadcast* (spreading a message to a group of processes), the scan operation, scatter (data items are distributed from one processor to all others) and gather (one processor gets data items from all others) operations. Some collective communication operations are presented in Fig. 2.11.

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

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

A typical parallel I/O scenario is that of multiple processors in a MIMD machine (Fig. 2.5b,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 to this bottleneck. DM architectures on the other hand, 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 performance problem of I/O lies in aggregating several requests to serve them efficiently. The well-known UNIX-like semantics of most file systems does not allow this transformation [72]. Indeed, the gain is even higher if the program explicitly gives information about collective I/O (parallel, logically synchronized I/O requests from a set of processors).

- **MPI derived data types.** are a portable mechanism to specify the memory layout of a data structure. They allow MPI functions to minimize communication overheads, and to compact non-contiguous data structures automatically. MPI-2 has extended the use of MPI data types from communication to parallel I/O. Since MPI-IO also offers collective and asynchronous I/O functions, there is plenty of room for optimizations.

Object-oriented Message-Passing with TPO++ Object-oriented programming is an important concept for sequential programming and will increasingly influence parallel programming in the future. The MPI-2 standard defines C++ language bindings for MPI-1 and MPI-2. However, the bindings do not provide enough concepts for real object-oriented message-passing programs. For example, interfaces are not type-safe, objects cannot be used as arguments for communication calls and the calls themselves were not simplified. TPO++ (Tübingen Parallel Objects) [38] is an object-oriented message-passing library written in C++ on top of MPI. The main design goals were:

- Integration of the STL (Standard Template Library).
- Capability of transmitting objects in a type-safe manner.
- Account for all recent C++ features like the usage of exceptions for error handling.
- Thread-safety, although this depends on the underlying MPI implementation.

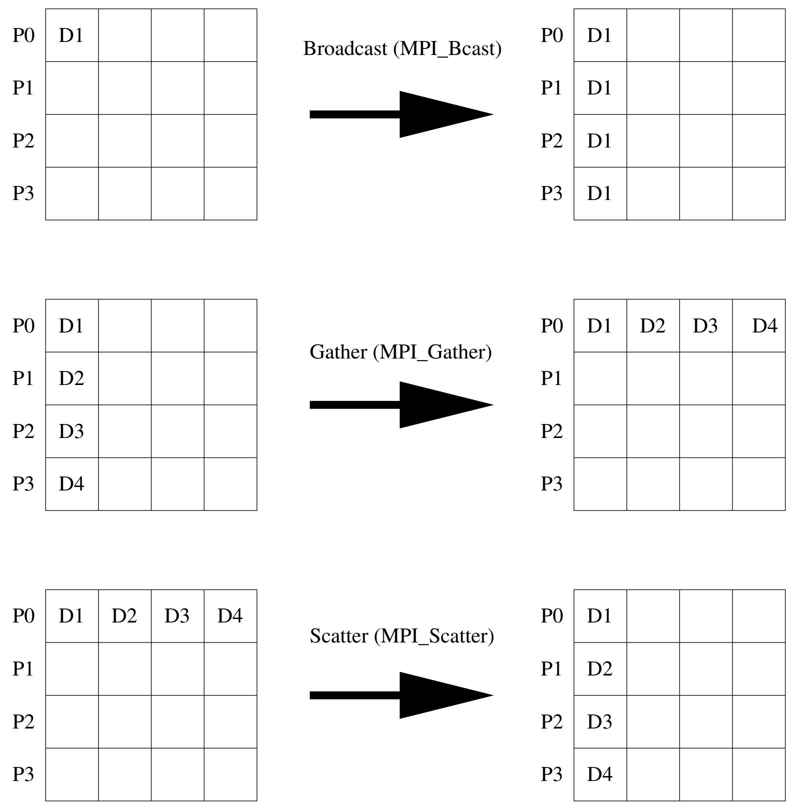


Figure 2.11: Examples for collective communication operations. Based on these operations, there are further operations where all processors are senders and receivers (Allgather, Allscatter, etc.).

Of course, all that is achieved without degenerating the communication and memory efficiency of MPI dramatically. For our experimental tests in Chapter 5 and 7, we used TPO++.

OpenMP

The *OpenMP-API* [61] is a standard for parallel shared-memory programming. Directives are a way to parameterize a specific compiler behavior. The directives are ignored if they are unknown to the compiler. OpenMP defines a set of program directives, with which it is possible to mark parallel regions in a sequential program without changing their semantics. While in C and C++ `#pragma` statements are used for the implementation of the directives, FORTRAN uses comments. It facilitates an incremental approach to the parallelization of sequential programs. The sequential part of the code is executed by one thread (master thread) that forks new threads as soon as a parallel region starts and joins them at the end of the parallel region (*fork-join model*). There are three types of directives[3]:

- **Parallelism/Work sharing** directives mark parallel regions in the program. A certain number of threads execute the code within a parallel region. A work sharing directive may appear within the parallel region that divides the computation among the threads. There are three different work sharing directives.

  (1) *for Construct*; the iterations of an associated loop can be executed in parallel and are thus distributed across the threads that already execute the parallel construct to which the *for construct* binds. Further it is possible to influence the distribution by the `schedule` clause. Among other possibilities, the iterations can be assigned statically , or dynamically. A static schedule divides the number of iterations into chunks of equal size. The chunks are statically assigned in a round-robin-fashion to the threads. In contrast, a dynamic schedule divides the iterations into a series of chunks of a given size. Each chunk is assigned to a thread waiting for an assignment. The thread executes its chunk and then waits for its next assignment. This is repeated until no chunk of iterations remains.

  ```
  #pragma omp for [clause[[,]clause] ...]
        for-loop
  ```

  (2) *sections Construct*; this defines a non-iterative work-sharing construct. A set of sections can be defined that are executed once by one thread of the parallel region.

  ```
  #pragma omp sections [clause[[,]clause] ...]
     {
     [#pragma omp section]
         structured block
     #pragma omp section
         structured block
     #pragma omp section
         structured block
     ...
     }
  ```

  (3) *single Construct*; this directive defines that a given structured block is executed only by one thread.

[3]We use the syntax for the directives as defined in the OpenMP specification for C/C++ [61].

```
#pragma omp single [clause[[,]clause] ...]
       structured block
```

- **Data environment** clauses control the sharing of program variables that are defined outside a parallel region. They are used within the declaration of the work-sharing directives explained above. The clauses are private, firstprivate, lastprivate, shared, default, reduction, copyin and copyprivate. We just explain three of them in more detail. The `private` clause requests each thread to create a new instance of the variable within the context of each thread. The variable can be modified by each thread without being visible to other threads. Variables defined inside a parallel region are implicitly private to each thread. In contrast, the `shared` clause defines that variables are really shared by all threads. Modifications on these variables are visible for the other threads. With the `reduction` clause, it is possible to compute a given operation in parallel for scalar variables. At the end of the region for which the reduction was specified, the original variable is updated with the value that results from the reduction made by all threads with the given operation.

- **Master and synchronization** directives are responsible for synchronized execution of several threads. Synchronization is necessary to avoid deadlocks and data inconsistencies. There is the master, critical, barrier, atomic, flush and ordered constuct. The `master` construct marks a structured block of code that is only executed by the master thread. A `critical` block is only executed by one thread at the same time. At a `barrier`, the threads stop their execution until all threads reach this point, too. With the `atomic` construct, it is possible to edit a shared variable without the risk of inconsistencies. The `flush` directive leads to a consistent view of the common data for all threads. Finally, the `ordered` construct can be used as clause in the *for* work-sharing directive and leads to an ordered execution of the iterations of the loop. Additionally to explicit synchronization calls, barrier is always called implicitly after a work-sharing directive.

- **Run-time library functions** The run-time library functions of OpenMP can be divided into two types. Several functions can be used to control the parallel execution, and lock functions for a synchronized access to shared data. An example for the first type is the `omp_get_thread_num` function. With this function, it is possible for each thread to request its own identity number. In dependence of this number, it is for example possible to execute different code blocks on different data like in MPI.

This section is only a brief overview of the possibilities for parallel programming with OpenMP. More detailed informations can be found in the actual specification [61].

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

2.4.2 Parallel-Hierarchical Programming

Hierarchical Optimizations for MPI

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

Shared-Memory Communication This approach improves the communication between processors that 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 that resides in the same node. If this is the case, the message will be delivered through shared-memory, otherwise over the network. Since the access cost to main memory is the significant factor of the performance of this inner-node communication, it is important to reduce the number of copies necessary to deliver the message.

In [71] optimizations of inter- and inner-node communication for a free MPI implementation called MPICH [37, 36], that works on PC-based SMP clusters, are presented. Initially, the library needs two shared-memory copy operations to perform the inner-node communication using features of a UNIX kernel. 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. By building a kernel primitive, that writes the message directly into the receiver's memory, only one copy operation is necessary.

In order to test the library, the authors made experiments using the *NAS Parallel Benchmark 2.3* (NPB) [7]. The NPB is a set of 8 programs designed to help evaluate the performance of parallel supercomputers.

The authors compared the results of an SMP cluster with that of a 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 of the advanced inner-node communication. In general, the latency for each process in a SMP cluster is higher than in a 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. If an application has to synchronize a lot, the advantage of the faster inner-node communication is wasted. The programs of the NPB are examples of such applications.

Though the point-to-point communication between processors in the same node can be improved by this approach, it is not guaranteed to improve the overall performance compared to that of a UP-cluster. Indeed, the programmer is not forced to consider the SMP cluster architecture during the design of an application. The SMP cluster can be treated like a non-hierarchic MPP machine, and the efficiency of the resulting program only depends on the MPI implementation and on the general program structure.

Threads Only MPI — The *threads only MPI* (*TOMPI*) [27] is an MPI implementation for uni-processor and SMP workstations. The idea is to make the development of MPI programs on workstations less time-consuming. It is inefficient to use standard MPI implementations, because for the sake of portability they use UNIX processes and UNIX domain sockets for interprocess communication. Both methods involve a large and unneeded overhead on a single workstation. An implementation using threads and shared-memory would perform much better. TOMPI rewrites an MPI program using a source code translator. The result is a program, using multiple threads on an SMP node or workstation. This approach seems to have the potential to be more efficient than the one based on shared-memory communication, because it does not only use a faster shared-memory communication within the nodes, it even avoids large memory overhead using processes. 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. Even though there is no implementation of the approach for SMP clusters yet, an automatic conversion of processes to threads is an interesting opportunity for this kind of architectures.

Distributed Shared-Memory Programming with OpenMP

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

Software distributed shared-memory *SDSM* systems provide a global address space for physically distributed-memory machines via a software library. We can translate OpenMP directives into appropriate calls to the

SDSM system. An example of this approach is described in [43]. The result of the source-to-source translation is a standard C/C++ or FORTRAN program , which can be compiled and linked with the SDSM system TreadMarks [2]. Further, the TreadMarks system is modified to exploit the hardware shared-memory within the SMP nodes. Experimental testing with several algorithms showed that the performance of the modified SDSM system got much better compared to the version not using shared-memory inside a node. However, compared to equivalent MPI implementations the performance is still worse. The speedups obtained were only within 7-30% of the speed-ups achieved by the MPI versions. Performance suffers too much from coherence-maintenance network traffic. Further, SDSM systems do not exploit application specific data access patterns, because communication in shared-memory is unknown until run time. Hence, more promising is the *compiler directed SDSM* approach [63], which is a two-step optimization. In a first step, the OpenMP compiler inserts memory coherence code, called *check code* primitives, to keep the node-distributed-memory consistent. There are three types of check codes, two of them ensure that the data is valid before a read or write of shared data, the third is responsible to inform the other nodes that data has been changed after a shared write. In the second step, the compiler analyzes parallel regions in order to optimize communication and synchronization by removing unnecessary check codes. The following optimization strategies are applied:

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

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

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

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

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

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

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

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

- **Data-parallel communication optimization.** Besides the reduction of check codes, it is possible to improve the program by using data-parallel compilation techniques. For example, the compiler should determine data mappings of arrays that are accessed contiguously in a loop in the way that the iterations of the loop can be done locally on the nodes where the data is stored. Since the number of threads is not known during compile time, the compiler has to insert calls to data mapping runtime library primitives that determine the loop bounds and data that must be communicated. The check codes can be removed because the data is stored locally on each node.
- **Collective communication optimization.** Inter-node communication is necessary to implement a reduction operation on variables defined in the data scope attribute of a parallel region. It can be performed efficiently using a collective communication library. The execution starts after the local reduction at the end of parallel regions or after work-sharing directives.

Distributed OpenMP — A different approach to adapt OpenMP to SMP clusters is suggested in [55]. The authors propose the *distributed OpenMP*. This extension of OpenMP with data locality features provides a set of new directives, library routines and environment variables. One data-distribution extensions is the `distribute` directive with which it is possible to partition an array over the node memories. For performance reasons, the threads should work on local array elements. Hence, the user must distribute the data in order to minimize remote data accesses. Another proposed extension is the `on home` directive in a parallel region. With this directive, it is possible to perform a parallel loop over a distributed array without redistributing the array. The threads of a node perform the iterations for the array elements that reside in their local memory. Further extensions are library routines and environment variables that provide specific numbers of the run-time instance of the SMP cluster, like for example the number of involved nodes or processors per node. Disadvantages are that programs get more complex, and the user still has to take care about efficient data decomposition. Therefore, after adding the new directives to an OpenMP program, the user probably has to do performance tuning again.

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 of SMP clusters, therefore, the model has potential to produce programs with significant performance improvement. However, 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 that 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 [16], there are two approaches to incorporate OpenMP directives into MPI programs, the fine-grain and the coarse-grain approach.

Fine-Grain Parallelization The hybrid *fine-grain parallelization* is done incrementally. The computational part of 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 cannot be parallelized directly. Nevertheless, if these loop-nests contribute significantly to the global execution time, the developer can try to transform them 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 an 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 an MPI process. For threads, there are several issues to consider:

- The **data distribution between the threads** is different from that of MPI processes. Because of the shared-memory, it is only necessary to calculate the bounds of the arrays for each thread. There has to be a mapping from array regions to threads.

- The **work distribution between the threads** is made according to the data distribution. Instead of an automatic distribution of the iterations, some calculations of the loop boundaries depending of the thread number define the schedule.

- The **coordination of the threads** means managing critical sections by either the usage of OpenMP directives, like `MASTER` or thread library calls like `omp_get_thread_num()`, to construct conditional statements.

- **Communication** is still done by only one thread.

As far as we know, until now, there has been no work on the coarse-grain approach. If we remember 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 that has to be incorporated in a 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 — Besides knowing how to develop hybrid programs for SMP clusters, it is necessary to look at the achieved performance of the programs. In [16, 17, 18, 22] investigations to measure performance of fine-grain hybrid programs are presented. They try to measure which programming model performs better on an SMP cluster by comparing the performance achieved by a hybrid and a pure MPI version of NPB. 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 of 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 papers, the comparison between the two kinds of models for SMP clusters shows no general advantage of one over the other. Depending on the characteristics of the application, some benchmarks perform better with the hybrid version; others perform better with the pure MPI version. The following aspects have influence on the performance of the models.

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

- 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 applications the pure MPI approach is worse.

- **Memory access patterns**. The memory access patterns are different for the two models. Whereas MPI allows expressing 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.

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

SIMPLE Model

The significant difference between *SIMPLE* [5] and the manual hybrid-programming approach above lies in the provided primitives for communication and computation.

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

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

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

- **Memory management.** The allocation of memory from the heap can be done by each thread using the `node_malloc` primitive. As an argument, the primitive gets the number of bytes needed and it returns a pointer to the memory address. In order to free memory space the threads can use the `node_free` primitive.

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

High Performance FORTRAN for Hybrid-Parallel Programming with HPF

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

One exception is presented in [10]. Therein, HPF is extended with the concept of *processor mappings* and the concept of *hierarchical data mappings*. With these two concepts, it is possible for the programmer to consider the hierarchical structure of SMP clusters. The VFC compiler [9] is extended in the way that it creates *fine-grain hybrid programs* using MPI and OpenMP out of an extended HPF program.

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

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

- **Intrinsic functions.** Two new functions are provided. The first one returns the number of nodes and the second one returns the number of processors in the SMP clusters. The functions are provided in order to support abstract node arrays whose sizes are determined upon start of a program.

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

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

block is a standard HPF distribution format and divides the concerned dimension into equal parts with respect to the distribution target. The asterisk defines that the whole dimension of the array will be mapped to the target elements, see also Fig. 2.12.

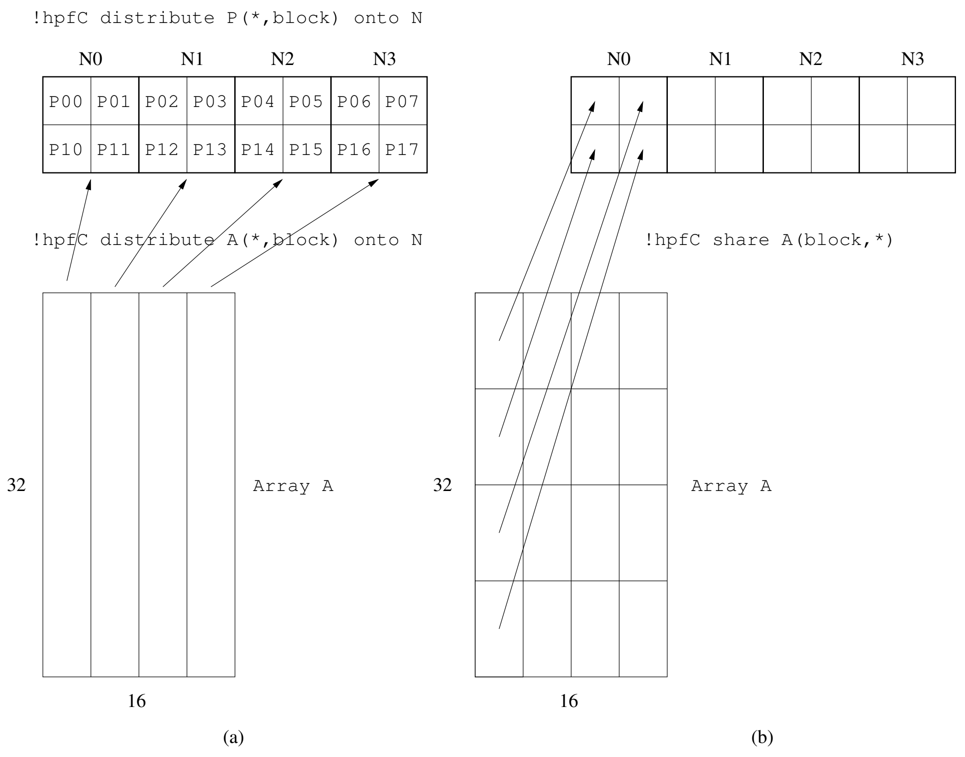


Figure 2.12: In (a) the partitioning of the processor array P(2,8) onto the node array N(4) is provided. Further, the decomposition of the array A onto the node array is shown. Vertical slices from array A of equal size are assigned to each node. In (b) it is shown how the share primitive divides the vertical slice onto the processor in each node. Only the mapping for node N0 is shown as an example.

KeLP2 Model

The *Kernel Lattice Parallelism 2 Model (KeLP2)* [4] is a runtime infrastructure in C++ that implements a methodology for implementing (irregular) block-structured numerical applications on SMP clusters. KeLP2 provides mechanisms to coordinate data decomposition, data motion and parallel control flow similar to HPF. As HPF, 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. The programmer can do the decomposition best and KeLP2 provides a framework to construct partitioning libraries. However, in comparison to HPF, it is possible to overlap computation and communication. In contrast to KeLP2, SIMPLE provides lower-level primitives, does not support data-decomposition, and it does not overlap communication and computation. Nevertheless, it is not as narrow in scope as KeLP2, concerning the application domain. KeLP2 supports three levels of control, the collective level (SMP cluster), the node-level, and the processor-level. Programs express parallelism at the node- and processor-level while communication takes place in the collective- and node-level. Both levels, the collective- and the node-level, have their own data-layout and -motion. The KeLP2 programming abstractions help to manage this. There are three classes of KeLP2 programming abstractions.

1. The *Meta-Data* represents the abstract structure of some facet of the calculation. It describes the data decomposition and the communication patterns.

2. *Instantiators* execute the program according to the information contained in the meta-data.

3. The primitives for *parallel control flow* are iterators, which iterate over all nodes or over all processors of a specified node.

KeLP2 enables a parallel specification that is less dependent on the implementation.

2.5 Summary

In Section 2.1, we showed that there is a trend to hierarchical parallel architectures and especially to SMP clusters. Hence, it is clear that there is a need for theoretical cost models, programming models and new software tools to exploit these hierarchical architectures.

Concerning the cost-models, we showed the evolution from PRAM to extensions for the BSP model. The concept of the BSP model is based on a more abstract definition of communication cost, and it introduces the superstep concept. Although this was a big advance in modeling parallel computers, it does not consider hierarchical structures of parallel machines. Several approaches try to make the BSP model more precise and accurate for real machines. In this context, we presented the BSP* and the D-BSP model.

The D-BSP model might be a candidate for modeling SMP Clusters. However, although it does support hierarchical structures with the concept of BSP sub-machine, it does not support the conceptual difference between shared-memory and message-passing communication needed for

SMP Clusters. Further, it seems to be also very difficult to find an appropriate and practically accepted programming model for D-BSP.

Concerning programming models, we introduced the two main streams of parallel programming models. OpenMP is the de facto standard for shared-memory programming and MPI is used for message-passing-based programming. We presented optimized versions for SMP clusters of both models. It is obvious that the shared-memory-based approach with OpenMP has performance drawbacks, because of the underlying distributed memory architecture of SMP clusters.

As we saw in Section 2.4.2, most of the efforts in parallel software development are spent in maintaining existing programming models and optimizing them for new architectures.

In general, this approach is not efficient enough 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 and algorithm developers 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. Hence, it is better to give the programmer or algorithm developer the control over the hierarchy. Based on these two programming models a new model was suggested that is a combination of both. OpenMP (or threads) is used for the parallel computation within the nodes and MPI is used as the communication library between the nodes. The idea is to use the respective model in that part of the machine where the physical architecture is closer to it. The hybrid model is the only parallel and hierarchical programming model accepted in 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. On the other side, using this model might be a bridge from a cost model to SMP Clusters and it seems to have the most potential concerning programs performance.

Therefore, what we want to do in the following is to examine if there are possibilities or methods for algorithmic optimizations, which exploit the hierarchical architecture. We start by formulating a cost model that reflects hierarchical SMP clusters and the hybrid-programming model. The cost model is the basis for the optimization and analysis of algorithms throughout this work.

Chapter 3

A Computational Cost Model for Hierarchical SMP Clusters

As we have explained in Chapter 2, there is a trend to a small number of classes of parallel architectures. Since the architectures of each class are completely different, like for example the classes of PC-based SMP clusters and vector-processors, it seems to make no sense to formulate a common cost model for all kinds of supercomputers, because the characteristics are too different. In contrast, we propose the strategy to take the parallel bridging model (PBM) that fits best to the class of the target architecture and extend it with the characteristics that the PBM does not reflect. Consequently, there will be separate models for the few classes of architectures. In the following, we will define a model for the class of SMP clusters. An advantage of starting on top of PBMs is that there are many algorithms, which can be adapted to the extended model.

3.1 Design Decisions for the Model

The main issues for the development of computational models in the past were generality, reality and simplicity. The aim was to cover all kinds of parallel computers and to predict the running time of algorithms as exact as possible. Despite of these high demands, the use of the model should be as simple as possible. This led more or less to a discussion about which and how many parameters are necessary for a general-purpose cost model.

It is obvious that building a general-purpose cost model means to make compromises. Special features of certain machines have to be ignored in order to be general enough. On the other hand, very important features have to be considered although this might decrease the set of covered machines.

The usage of a general-purpose cost model comprises the risk of not considering relevant elements. Hence, the algorithm analysis may lead to results that cannot be confirmed practically on the target machine.

This danger is given for SMP clusters because there are several characteristics that are not considered in most PBMs.

- There are different communication styles concerning communication within the SMP node and between SMP nodes. It is more expensive to communicate over the network than by shared-memory.

- Looking at the basic operations of the inter- and inner-node communication differences can be recognized. While between SMP nodes the point-to-point communication is clearly the basic operation, within an SMP node a one-to-all operation is the more natural basic operation, because of the shared-memory.

- Concerning the communication over the network, there might be even differences between the costs of communication operations depending on the number of network hierarchies between the sending and receiving processor.

- Several processors have direct access to the same memory space. Hence, redundant data usage in an SMP-node may also lead to improvements. Using less data per processor or SMP node improves the process performance, because memory is a critical resource.

Another argument for a more detailed model is given by the fact that SMP clusters build a big and increasing fraction of the family of supercomputers and therefore can be regarded as an own class.

Together with the more detailed model, there should be a methodology to transfer well-known and practically proven algorithms from general-purpose cost models. This does not only lead to a solid base of algorithms for the new model, it does also improve the understanding of the new machine and creates ideas for the design of more efficient algorithms. However, in this chapter, we concentrate on the model as a quantitative base for algorithm design. In Chapter 4, we address to general design methods. In the following, we design a model for SMP clusters based on parameters and ideas of more general cost models like BSP, and extend it by the main features of actual SMP clusters. In the following, we will outline and shortly discuss the main features:

- **Inner- and inter-node communication.** The most important characteristic of SMP clusters is the division into shared-memory and network connected processors. The difference between an inner- and an inter-node communication has to be considered in the model. The inner-node communication should not be based on point-to-point connections because the shared-memory should be exploited for collective operations.

- **Hierarchical inter-node connection.** With the help of special fast network switches, it is possible to build huge clusters of SMP nodes. Generally, these switches are connected in the way that they build a tree-like hierarchical network. The more switches are involved in a message-passing operation the more overhead is involved for getting a way through each switch. Hence, for each switch we pass, a constant overhead of time should be added to the latency while we assume only one bandwidth for the network.

  On the other hand, due to Metacomputing and Grid computing technologies further levels of hierarchy have to be considered that provide different bandwidths. Hence, in order to be general enough for all situations the model should define a level dependent latency and bandwidth. Unfortunately, this increases the amount of parameters, and decreases the ease of use of the model. One assumption of the network hierarchy is that the higher the level of communication within the hierarchy, the higher the latency and the lower the bandwidth. Due to this observation, it is enough to consider different latencies for each level, but only one bandwidth. The model rewards algorithms that communicate in lower levels by a lower latency for communication. The faster bandwidth of these levels is implicitly rewarded. Hence, for algorithm design it is sufficient that the model considers different latencies and we can reduce the set of parameters. Despite of that, bandwidth has to be considered too, in order to assess message lengths for network communication and memory copy operations. Thus, we can compare the communication behavior of algorithms by their over-all latency, their over-all time for network bandwidth and their over-all time for shared-memory bandwidth.

- **Asynchronous Communication.** The communication hardware of many computers is capable to send several messages one after the other without having to wait for the end of the transition of the preceding send operation. This can be exploited by communication libraries like MPI, and hence should be considered in the model.

- **Homogeneity of architecture.** Due to the high scalability of SMP clusters, it is foreseeable that SMP clusters will not stay homogeneous. Despite of that we do not think it is necessary to incorporate heterogeneity in the model. The algorithms are developed in SPMD style. Therefore, heterogeneity of the SMP nodes will have influence on the data distribution between the nodes. This can be done later according to the performance ranking of the nodes, and will not influence the algorithm.

3.2 κNUMA Model

Under consideration of these features, we define the κNUMA model, a computational model for SMP clusters. The κ in the name stands for number of the levels in the hierarchy of the interconnection network between the SMP nodes. The second part of the name, NUMA, accounts for the non-uniform access that exists because the local memories are separates by the network hierarchy.

The structure of a κNUMA-machine resembles a complete tree. The leaves of the tree are the processors and the inner nodes correspond to the levels of the network. The inner nodes are connected by edges that correspond to inter-network interfaces or switches. The tree of a κNUMA-machine has the height $\kappa + 1$ and the degree of all inner nodes with the same height is equal. Fig. 3.1 is an example for a 2NUMA-machine. The computational cost model will be described by its parameters. There are parameters, which describe the κNUMA-architecture and others, which are necessary for modeling the κNUMA-communication.

3.2.1 The Set of κNUMA-Parameters

Architectural Parameters

- κ is the number of hierarchies in the network. Level 0 is the internal dynamic interconnection network between processors and memory banks. of an SMP node. Hence, the distance from a SMP node to the root of the tree is κ and the distance from a processor to the root is $\kappa + 1$.

- $\alpha(l)$ is the number of the $(l-1)$NUMA sub-machines in level l (subtrees of height l) with $0 \leq l \leq \kappa$. $\alpha(0)$ is the number of processors in an SMP node. Thus, the numbers of processors p is not really necessary as a parameter, because $p = \prod_{k=0}^{\kappa} \alpha(k)$. All lNUMA sub-machines of the same level $l+1$ consist of the same number of $(l-1)$NUMA sub-machines. For example, a 0NUMA-machine is a single SMP computer.

Communication Parameters

- s_i is the latency. Every time during the transmission when the data changes the level of network (upward or downward in the tree), there is an overhead of at most s_i (latency), whereby $0 \leq i \leq \kappa$. To be more precise, s_0 is the latency for local memory access and s_i, with $1 \leq i \leq \kappa$, represents the overhead incurred by the transmission through the levels of the network (switches).

- g_{local} is the local bandwidth in bytes per unit of time (Bytes/sec) for transferring data within the SMP node.

- g_{global} is the global bandwidth in bytes per unit of time (Bytes/sec) for transferring data through the levels of the network.

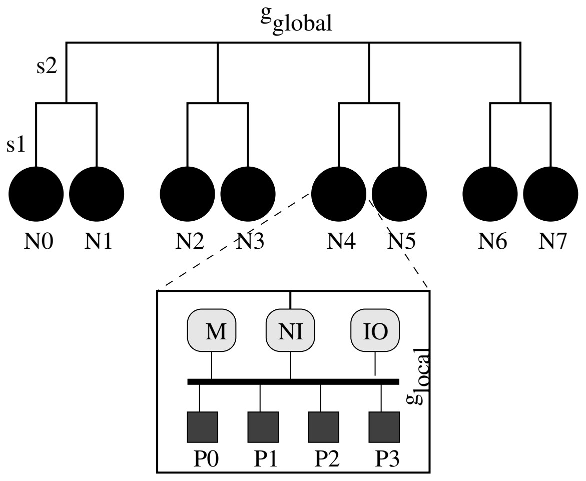


Figure 3.1: Example for a 2NUMA-machine, where $\alpha(2) = 4, \alpha(1) = 2, \alpha(0) = 4$.

3.2.2 Execution of Parallel Algorithms

The model behaves similar to the BSP model of Valiant [76]. Algorithms consist of consecutive supersteps. To simplify presentation, every superstep is either purely computation on local data or purely communication. After each superstep, there is a *virtual* barrier synchronization between the processors. In practice, barrier synchronization is very expensive and should be avoided if possible. Hence, the barrier synchronization in the model is defined implicitly. No processor can continue his execution without finishing the communication superstep, because it needs the incoming data. In this sense, the barrier is virtual, because some processors might get the data earlier than others might and can immediately start with the next computational superstep.

The total cost of a parallel algorithm can be calculated by aggregating the costs of all supersteps required. The costs for computation supersteps are estimated using asymptotic analysis like in the other computational cost models. In the following, the costs for communication supersteps are presented.

Communication Operations

The costs for a communication superstep are the maximum of all costs of communication operations performed by the processors in that superstep. The communication between two processors can be divided into two parts. There is the inter-node communication where both processors are situated in different SMP nodes, and there is the inner-node communication where the engaged processors are in the same SMP node.

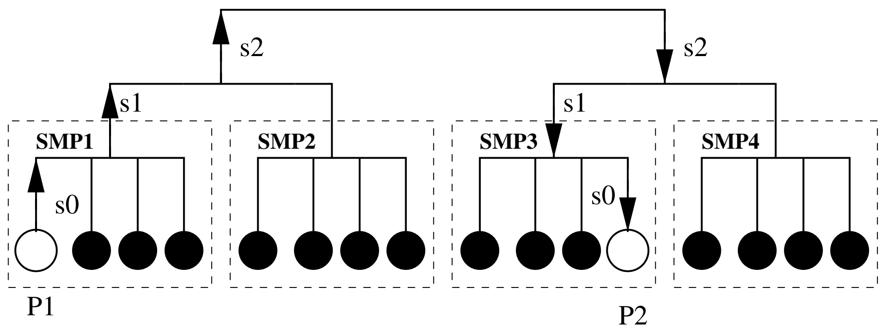


Figure 3.2: Inter-node communication in the κNUMA model, where $\kappa = 2 = l$.

Inter-node Communication Let P1 be the sending and P2 be the receiving processor of data of size L. P1 and P2 are situated in different SMP nodes. The highest level of network the data has to traverse is l ($1 \leq l \leq \kappa$). The communication is illustrated in Fig. 3.2. The time needed for the operation is charged as

$$T_{put} := \underbrace{s_0 + \frac{L}{g_{global}}}_{T_{block}} + 2\sum_{i=1}^{l} s_i + s_0 = \frac{L}{g_{global}} + 2\sum_{i=0}^{l} s_i \qquad (3.1)$$

This time consists of the latency of the local memory of P1, the writing of the data in the network, the latency of the network itself and the latency of the access to the local memory of the remote processor P2. T_{block} is the time after which P1 can continue doing other work if the operation is non-blocking. After T_{block} P1 can start the next operation. Hence, T_{block} is comparable with the gap parameter g of the LogP model, although it depends on the message length.

Inner-node Communication The basic idea to model the communication between two processors in the same SMP node is as follows. On each SMP node, there is one process with multiple threads (one thread for each processor). The threads have private data as well as shared data. A thread can communicate with the others by making shared data out of private data. The other threads make local copies of the new shared data for their use. Shared-memory models like the QSM model [34] that was briefly described in Section 2.3.3 on page 26, typically divide the memory into a private and a shared part. Communication can be done in one-to-all fashion and consists of two internal steps. First, the sending processors write their data to the shared-memory. Second, all processors make local copies of the new shared data, which they need, see Fig. 3.3.

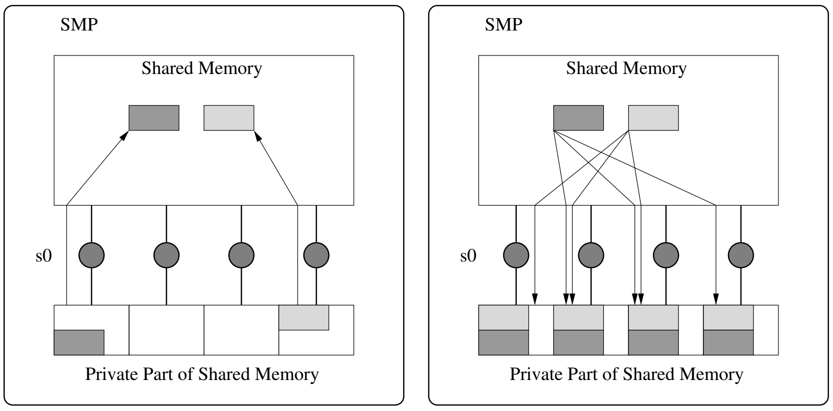


Figure 3.3: The two steps of an inner-node communication operation in the κNUMA model.

Let W_i be the size of data which will be written from processor p_i to shared-memory and R_i the size of data read by processor p_i in the second step. The runtime for the operation in one SMP node, where multiple processors may communicate, can be calculated as follows.

$$\underbrace{s_0 + \frac{\max_{1\leq i\leq\alpha(0)} W_i}{g_{local}}}_{\text{new shared data}} + \underbrace{\frac{\max_{1\leq i\leq\alpha(0)} R_i}{g_{local}} + s_0}_{\text{parallel copy}} \tag{3.2}$$

3.2.3 One-to-All Broadcast Problem

In order to introduce the model, we will analyze the casual broadcast problem and the *personalized broadcast* problem, which is the same as the scatter operation in MPI, see Fig. 2.11. In the latter problem, one processor sends $(p-1)$ different messages each of length M to the other $(p-1)$ processors, while in the former problem all $(p-1)$ processors will receive the same message. Each processor gets exactly one message.

Looking at the BSP model, it is obvious that the problem can be solved in $T = \log_d p$ supersteps by executing a logical d-ary tree. In each supersteps, d messages are sent by the involved processors until all processors received a message. The total cost for the broadcast in the BSP model is $(dg + L)\log_d p$. The choice of d is very important and depends on the capability of the router, the number of processors and the ratio between g and L in the BSP machine. Further, there is no difference concerning the communication costs whether we perform the casual or the personalized broadcast, because the BSP model does not account for message lengths.

We are interested how the broadcast algorithm can be adapted to a κNUMA machine. As in the BSP model, the choice of d depends on the actual values of the parameters that are different in each system. Thus, we cannot determine a general best value for d, however, we can show possibilities for the adaptation of the algorithm independent of d. Hence, in the following, we will present, analyze and compare two different algorithms for the problems that represent an extreme of the BSP approach. The first one is called *Direct 1-level broadcast* which is derived from the BSP-approach by setting $d = p$. Additionally, by setting $d = p$, we can show the use of asynchronous communication operations best. The second algorithm is called *κ-level broadcast*. Here, the broadcast is adapted to the architecture of the SMP cluster by applying the Direct 1-level broadcast algorithm to each level of the hierarchy. In both descriptions, P_i is the initiating processor of the broadcast. Since we are not only interested in the hierarchical structure, but also in the difference between inner- and inter-node communication, we assume that $\alpha(0) > 1$.

Direct 1-level broadcast

Processor P_i sends $(p-1)$ messages, each of length M, to the $(p-1)$ processors directly. Hence, with this algorithm we can solve the casual as well as the personalized broadcast problem. In both cases the cost for the algorithm is equal, because it does not matter if the messages for each processor are different or not.

Assuming the router of the BSP-machine is able to perform this operation in one superstep ($h \geq p$ for the h-relation of the BSP machine) and g is much smaller than L (synchronizing an SMP cluster over the network is expensive compared to sending p messages), this algorithm is optimal in the BSP-model, because only one communication step and thus, only one barrier synchronization with cost L is necessary. In the following, only *non-blocking* communication operations are used in the algorithm. Therefore, the times of the consecutive operations can overlap. It is clear that the order in which the messages are sent is decisive for the resulting runtime, because the distance to the receiving processor varies for each message. In the following, we present the two extreme running times subject to the order in which the messages are sent.

- In order to overlap the operations maximally, the messages with the largest distance should be sent first, and the messages with the smallest distance should be sent last. The optimal running time of the algorithm can be achieved if the latencies of all messages can be hidden by subsequent operations. A sample is illustrated in Fig. 3.4. The *best-case* runtime of the direct 1-level broadcast can be charged as:

$$\underbrace{(p-\alpha(0))(s_0+\frac{M}{g_{global}})}_{\text{Inter-node Communication}} + \underbrace{2s_0+\frac{\alpha(0)M}{g_{local}}}_{\text{Inner-node Communication}} = \tag{3.3}$$

$$= (p-\alpha(0)+2)s_0+\frac{(p-\alpha(0))M}{g_{global}}+\frac{\alpha(0)M}{g_{local}} \tag{3.4}$$

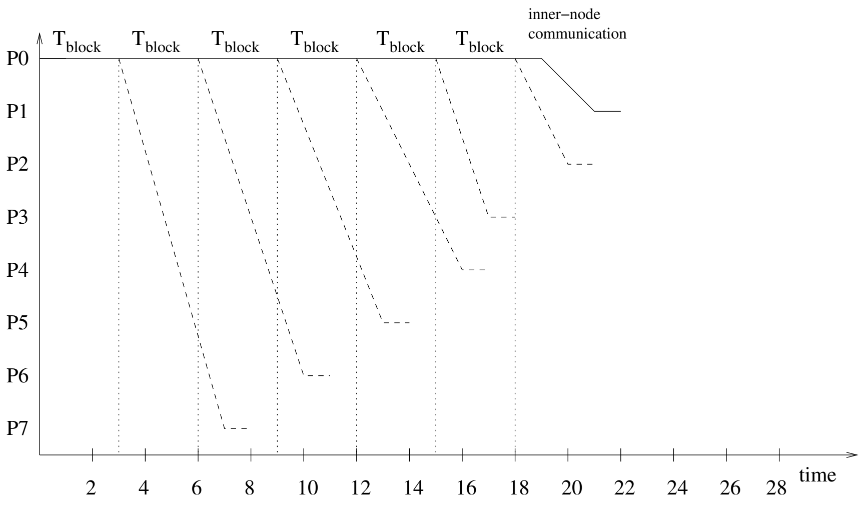


Figure 3.4: An example for the best-case of the direct 1-level broadcast algorithm. The target SMP clusters is depicted in Fig. 3.6, where $\alpha(0) = \alpha(1) = \alpha(2) = 2$. Further, we assume that $g_{local} = 2, g_{global} = 1, M = 2$ and $s_i = 1$ with $\forall i : 0 \leq i \leq \kappa$. P_0 is the initiating processor. The time needed for the broadcast is 22.

- The *worst-case* of the algorithm occurs if the order in which the messages are sent is vice versa. First, P_i sends the messages to the processors in the same SMP node, and then it sends the messages to the processors with ascending distance. Fig. 3.5 shows an example. The worst-case has, therefore, the following running time:

$$\underbrace{2s_0 + \frac{\alpha(0)M}{g_{local}}}_{\text{Inner-node Communication}} \underbrace{+ (p - \alpha(0))(s_0 + \frac{M}{g_{global}}) + 2\sum_{i=1}^{\kappa} s_i + s_0}_{\text{Inter-node Communication}} = \tag{3.5}$$

$$= (p - \alpha(0) + 3)s_0 + 2\sum_{i=1}^{\kappa} s_i + \frac{(p - \alpha(0))M}{g_{global}} + \frac{\alpha(0)M}{g_{local}} \tag{3.6}$$

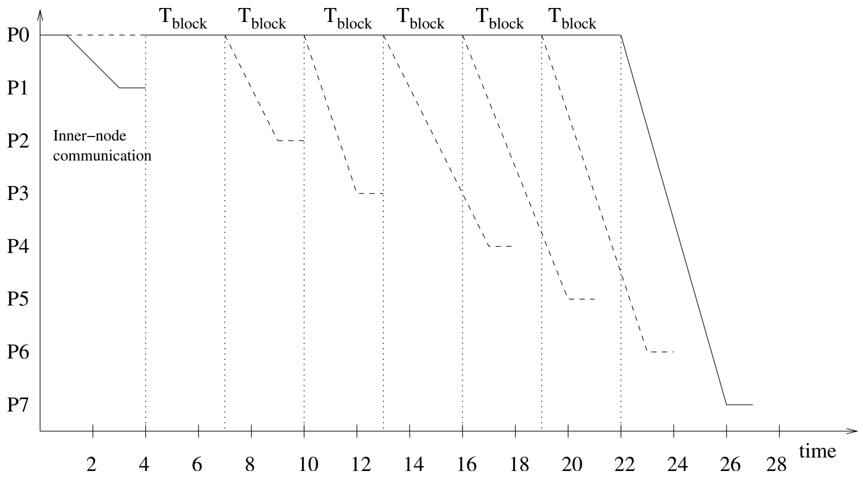


Figure 3.5: An example for the worst-case of the direct 1-level broadcast algorithm. The target SMP cluster is depicted in Fig. 3.6. Further, we assume the same parameter setting for g_{local}, g_{global}, M, s_i, $\alpha(i)$ as in Fig. 3.4. P_0 is the initiating processor. The time needed for the broadcast is 27.

As we can see from the analysis, the order in which the messages are sent is very important for the running time. Models like BSP do not consider this, but good implementations have to. The next algorithm tries to use a broadcast tree that has the same structure as the underlying κNUMA-machine.

κ-level broadcast

The algorithms consists of two phases. In the first phase, the messages for all processors in the same SMP node are sent to one of them using the following algorithm. The highest level of the network hierarchy consists of $\alpha(\kappa)$ $(\kappa-1)$NUMA sub-machines. In every sub-machine, we determine one processor which represents the whole sub-machine. P_i sends all messages for the processors of the sub-machines to its representative. P_i itself is the representative for its sub-machine. This procedure is the first superstep

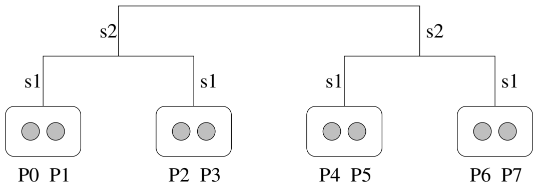


Figure 3.6: A sample architecture for the illustration of the broadcast algorithms of Fig. 3.4, 3.5, 3.7. The architectural parameters are set to $\kappa = 2$, $\alpha(2) = \alpha(1) = \alpha(0) = 2$.

of the algorithm. The next supersteps are executed in the same manner. Every representative does the same job for its sub-machines as P_i did in the first step. This procedure is repeated as long as there is one processor in each SMP node, which has all the messages for the processors in the block. Phase 1 consists of κ supersteps.

Then in the second phase, the processor uses inner-node communication to broadcast the messages in the block. Hence, phase 2 consists of 1 superstep. Every processor has its message after phase 2. Phase 1 is similar to the broadcast algorithms presented in [49, 12]. In the following we show the analysis of the two phases.

Phase 1: κ *supersteps*

step 1 The sending of $\alpha(\kappa) - 1$ messages of length M costs:

$$(\alpha(\kappa) - 1)(s_0 + \tfrac{M}{g_{global}}) + 2\textstyle\sum_{k=1}^{\kappa} s_k + s_0$$

step 2 The sending of $\alpha(\kappa - 1) - 1$ messages of length M costs:

$$(\alpha(\kappa - 1) - 1)(s_0 + \tfrac{M}{g_{global}}) + 2\textstyle\sum_{k=1}^{\kappa-1} s_k + s_0$$

...

step κ The sending of $\alpha(1) - 1$ messages of length M costs:

$$(\alpha(1) - 1)(s_0 + \tfrac{M}{g_{global}}) + 2s_1 + s_0$$

If we aggregate the costs of the κ steps, we get:

$$s_0 \sum_{i=1}^{\kappa} (\alpha(i) - 1) + \sum_{i=1}^{\kappa} \left((\alpha(i) - 1) \frac{M}{g_{global}} \right) + 2 \sum_{i=1}^{\kappa} \sum_{j=1}^{i} s_j + \kappa s_0 \qquad (3.7)$$

Phase 2: 1 *superstep:* Distributing the message within the SMP node costs two copies in the shared-memory:

$$2s_0 + \frac{2M}{g_{local}} \tag{3.8}$$

The overall running time can be calculated by adding times for (3.7) and (3.8).

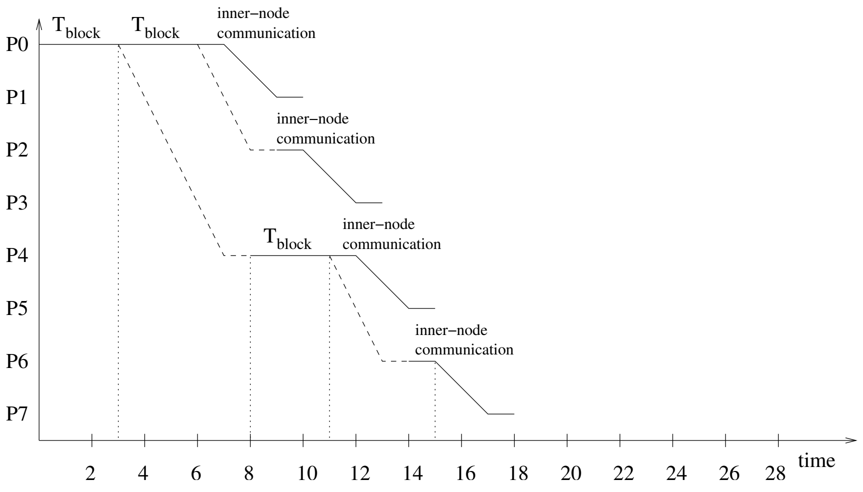


Figure 3.7: An example for the κ-level broadcast algorithm. Further, we assume that $g_{local} = 2$, $g_{global} = 1$, $M = 2$ and $s_i = 1$ with $\forall i : 0 \le i \le \kappa$. The time needed for the broadcast is only 18. Again, P0 is the initiating processor in the cluster of Fig. 3.6.

Comparison of Running Times

To simplify the comparison of running times, we make the following assumptions.

1. $\alpha(i) = \alpha$ with $\forall i : 0 \le i \le \kappa$. That means that we assume a symmetrical architecture. In particular $p = \prod_{i=0}^{\kappa} \alpha(i) = \alpha^{\kappa+1}$.
2. $s_0 = s_i$ with $\forall i : 0 \le i \le \kappa$. The costs for changing the levels in the network are the same as the latency of the local memory.

According to this assumptions, we can simplify the formula 3.7 by exchanging s_i with the constant s_0 and $\alpha(i)$ with the constant α. These modifications helps to eliminate the nested sums.

$$s_0 \sum_{i=1}^{\kappa} (\alpha(i) - 1) + \sum_{i=1}^{\kappa} \left((\alpha(i) - 1) \frac{M}{g_{global}} \right) + 2 \sum_{i=1}^{\kappa} \sum_{j=1}^{i} s_j + \kappa s_0 =$$

$$
\begin{aligned}
&= \kappa(\alpha-1)s_0 + \frac{(\alpha-1)M}{g_{\mathrm{global}}}\sum_{i=1}^{\kappa} 1 + 2s_0\sum_{i=1}^{\kappa}\sum_{j=1}^{i} 1 + \kappa s_0 \\
&= \kappa(\alpha-1)s_0 + \frac{(\alpha-1)M}{g_{\mathrm{global}}}\kappa + 2s_0\sum_{i=1}^{\kappa} i + \kappa s_0 \\
&= \kappa(\alpha-1)s_0 + \frac{\kappa(\alpha-1)M}{g_{\mathrm{global}}} + 2s_0\frac{\kappa(\kappa+1)}{2} + \kappa s_0 \\
&= (\kappa(\alpha-1) + \kappa + \kappa(\kappa+1))s_0 + \kappa(\alpha-1)\frac{M}{g_{\mathrm{global}}} \\
&= (\kappa\alpha + \kappa(\kappa+1))s_0 + \kappa(\alpha-1)\frac{M}{g_{\mathrm{global}}} \qquad (3.9)
\end{aligned}
$$

Additionally, we add the result of 3.8 (inner-node broadcast) to 3.9 and divide the formula into the categories latency (s_0), local bandwidth (M/g_{local}) and global bandwidth (M/g_{global}). The results are illustrated in Table 3.1. As we can see in table, concerning the local bandwidth, both algorithms are equal. The κ-level algorithm is clearly better concerning the global bandwidth. When we consider the latency, we can remark that the κ-level broadcast gets better for a more distinct κNUMA architecture. The higher the hierarchy of the network, the bigger is the difference between the running times.

algorithm	latency s_0	M/g_{local}	M/g_{global}
κ-level	$\kappa\alpha + \kappa(\kappa+1) + 2$	2	$\kappa(\alpha-1)$
Direct 1-level (best)	$\alpha^{\kappa+1} - \alpha + 2$	2	$\alpha^{\kappa+1} - \alpha$
Direct 1-level (worst)	$\alpha^{\kappa+1} - \alpha + 2\kappa + 3$	2	$\alpha^{\kappa+1} - \alpha$

Table 3.1: Running times of the casual broadcast algorithms divided into the categories latency (s_0), local bandwidth (M/g_{local}) and global bandwidth (M/g_{global})

κ-level personalized broadcast

In the following, we show the analysis of the κ-level algorithm for the personalized broadcast problem. The main difference to the analysis of the casual broadcast is that in each step the length of the messages vary, because the representative processor of a sub-machine gets all messages for all processors it presents. Hence with each step, the lengths of the messages decrease until they have length M in the last step. We will see that

the algorithm has the same costs for latency and local bandwidth as in for the casual broadcast, but it has a higher cost for global bandwidth.
Phase 1: κ *supersteps*

step 1 The sending of $\alpha(\kappa)-1$ messages of length $M\prod_{k=0}^{\kappa-1}\alpha(k)$ costs:

$$(\alpha(\kappa)-1)(s_0+\tfrac{M}{g_{global}}\textstyle\prod_{k=0}^{\kappa-1}\alpha(k))+2\sum_{k=1}^{\kappa}s_k+s_0$$

step 2 The sending of $\alpha(\kappa-1)-1$ messages of length $M\prod_{k=0}^{\kappa-2}\alpha(k)$ costs:

$$(\alpha(\kappa-1)-1)(s_0+\tfrac{M}{g_{global}}\textstyle\prod_{k=0}^{\kappa-2}\alpha(k))+2\sum_{k=1}^{\kappa-1}s_k+s_0$$

...

step κ The sending of $\alpha(1)-1$ messages of length $M\alpha(0)$ costs:

$$(\alpha(1)-1)(s_0+\tfrac{M}{g_{global}}\alpha(0))+2s_1+s_0$$

If we aggregate the costs of the κ steps, we get:

$$s_0\sum_{i=1}^{\kappa}(\alpha(i)-1)+\sum_{i=1}^{\kappa}\left((\alpha(i)-1)\frac{M}{g_{global}}\prod_{j=0}^{i-1}\alpha(j)\right)+2\sum_{i=1}^{\kappa}\sum_{j=1}^{i}s_j+\kappa s_0 \quad (3.10)$$

Phase 2: 1 *superstep:* Distributing the messages within the SMP node costs:

$$2s_0+\frac{\alpha(0)M}{g_{local}} \quad (3.11)$$

The overall running time can be calculated by adding times for (3.10) and (3.11). Again, according to the assumptions above, we can simplify the formula 3.10.

$$\begin{aligned}
&s_0\sum_{i=1}^{\kappa}(\alpha(i)-1)+\sum_{i=1}^{\kappa}\left((\alpha(i)-1)\frac{M}{g_{global}}\prod_{j=0}^{i-1}\alpha(j)\right)+2\sum_{i=1}^{\kappa}\sum_{j=1}^{i}s_j+\kappa s_0=\\
&=\kappa(\alpha-1)s_0+\frac{(\alpha-1)M}{g_{global}}\sum_{i=1}^{\kappa}\alpha^i+2s_0\sum_{i=1}^{\kappa}i+\kappa s_0\\
&=(\kappa(\alpha-1)+\kappa+\kappa(\kappa+1))s_0+\left(\sum_{i=1}^{\kappa}\alpha^{i+1}-\sum_{i=1}^{\kappa}\alpha^i\right)\frac{M}{g_{global}}\\
&=(\kappa\alpha+\kappa(\kappa+1))s_0+(\alpha^{\kappa+1}-\alpha)\frac{M}{g_{global}} \quad (3.12)
\end{aligned}$$

Again, we add the value of 3.11 (inner-node personalized broadcast) to 3.12 and divide the formula into the categories latency (s_0), local bandwidth (M/g_{local}) and global bandwidth (M/g_{global}). The results are illustrated in Table 3.1. As we can see in the table, concerning the local and global bandwidth, both algorithms are equal. When we consider the latency, we can remark that the κ-level broadcast gets better for a more distinct κNUMA architecture. The higher the hierarchy of the network, the bigger is the difference between the running times.

algorithm	latency s_0	M/g_{local}	M/g_{global}
κ-level	$\kappa\alpha + \kappa(\kappa+1) + 2$	α	$\alpha^{\kappa+1} - \alpha$
Direct 1-level (best)	$\alpha^{\kappa+1} - \alpha + 2$	α	$\alpha^{\kappa+1} - \alpha$
Direct 1-level (worst)	$\alpha^{\kappa+1} - \alpha + 2\kappa + 3$	α	$\alpha^{\kappa+1} - \alpha$

Table 3.2: Running times of algorithms divided into the categories latency (s_0), local bandwidth (M/g_{local}) and global bandwidth (M/g_{global})

3.2.4 Remarks and Conclusions to the Broadcast Problem

- Basically, the BSP model suggests one algorithm for the one-to-all-broadcast problem. In this algorithm the messages are distributed by a balanced d-ary tree [76]. In fact, there are p possibilities to implement this algorithm by setting d in the range from 1 to p. Hence, it is up to the parameters for the real machine which instance of this algorithm is best. The BSP model only gives an estimate how the algorithm works, but a more accurate description can only be achieved by analyzing the problem on the target architecture.

  In order to present the κNUMA model and its possibilities, we decided to analyze a certain instance of this algorithm that uses $d = p$. With the help of the κNUMA model, it was possible to analyze the extremes of this algorithm First, we analyzed the algorithm's behavior by applying it directly, just considering the order in which the processors are addressed by the initiating processor. We showed that the running time of the algorithm depends on this order, and presented best- and worst-case behaviors.

  In a second step, we constructed a new algorithm called κ-level broadcast by applying the direct algorithm to each level of the κNUMA machine, and analyzed it on the model, too. The comparison of these different algorithms showed that the κ-level is the best for typical values of the κNUMA parameters.

Hence, this is not only an example how algorithms can be analyzed using the κNUMA model, it does also show, how more efficient algorithms for SMP clusters can be derived from BSP algorithms and the model served as a quantitative basis for a comparative algorithm analysis.

- We do not claim that the κ-level algorithm is the best broadcast algorithm for the κNUMA model in general, but we showed how it was constructed out of an BSP algorithm and that it performs better than this algorithm. In general, the direct broadcast algorithm is not the fastest despite of using asynchronous communication operations. However, the fastest algorithm could be constructed in the same way as the κ-level algorithm by choosing the right d-ary tree for each level of the κNUMA machine.

- We can retain that the BSP model makes no statement about the order in which the messages should be sent. Nevertheless, as we showed, this point influences the running time on this architecture very much, because of the possibility to overlap the execution of consecutive asynchronous communication operations. Hence, a good implementation should take care of the sending order.

- The BSP model does not differentiate between a casual and a personalized broadcast, because it just account for the number of messages and not for their lengths. But as we showed, this makes a difference for algorithms with tree-like communication patterns. The sizes of the messages for each level in the tree are different and hence, the communication costs for these messages are different.

3.3 Summary

In this chapter, we defined a computational cost model for hierarchical SMP clusters denoted as κNUMA model. The model is based on concepts of widely used general-purpose cost models, like BSP, but is more accurate concerning communication in SMP clusters. In contrast to other models, κNUMA supports the hybrid nature of SMP clusters by differentiating between inner- and inter-node communications. Further, κNUMA considers asynchronous communication operations that are widely supported by actual message-passing libraries and allow overlapping consecutive communication operations. As an example for the use of the model, we analyzed the casual and the personalized broadcast problem, which is the same as the scatter-operation in the MPI library.

We presented and analyzed an algorithm derived from a BSP algorithm for the problem and compared the theoretical results with each other. We showed that the running times of the algorithms could vary depending on the order in which they are sent. This is important for implementations and is not considered in the BSP model. On the base of this algorithm, we constructed a new broadcast algorithm by applying the BSP algorithm to each level of the hierarchy. The comparison of the results of the analysis showed that the new algorithm performs better.

Hence, algorithms, which might behave optimal in the BSP model, are not necessarily optimal for the κNUMA model if they are applied directly. The BSP model is too coarse and therefore may lead to inefficient algorithms for SMP clusters. Despite of that, BSP algorithms are a good base for the development of efficient algorithms for SMP clusters, and with the κNUMA model it is possible to make a comparative algorithm analysis.

The formulation of the κNUMA model includes some compromises. The aim was to define a model that is general, but also accurate enough to account for the characteristics, we want to research for SMP clusters. An even more general model could e. g. use different bandwidth rates for each level in the hierarchy. Our approach is coarser because we use the same bandwidth for the whole network (implicitly assuming that the bandwidth is dominated by the lowest in the network). However, despite of that, κNUMA rewards communication in lower levels of the network by level individual constant costs. Because of that, algorithms try to avoid communication operations over high levels and by that they simultaneously use higher bandwidth rates, because generally the lower the level, the higher the bandwidth.

Another more general approach is to consider the (memory) hierarchy within the SMP nodes (registers, caches, memory). The κNUMA model just defines costs for shared-memory communication and assumes that computation on local data considers the respective memory hierarchy of the node. A κNUMA algorithm consists of computation and communication steps. In the computation steps, each processor computes on local data. Hence, the behavior of this sequential computation could be analyzed using sequential memory hierarchy-aware models and is therefore out of scope for the parallel analysis.

Due to these compromises, the κNUMA model can be regarded as a special case of general heterogeneous hierarchical systems that reflects the very common class of homogeneous SMP clusters. On the other side, despite of this specialization, it is already difficult enough to analyze algorithms with the κNUMA model. Even for the comparably simple broadcast and scatter operation, large formula had to be mastered. Hence, in the following, the κNUMA model can be also regarded as a super-model for the class of SMP clusters. It seems to be rational to reduce the model in dependence of the considered problem and the underlying architecture. Possibilities to improve the ease of use of the model are the fixing or reducing of parameters.

Another problem is that we have not verified the theoretical results in practice, yet. Thus, we will address to more practical examples for special cases of the κNUMA model in other chapters. However, before we can do this, it is necessary to deal with the design of parallel algorithms. The presented broadcast algorithms were developed straightforward and without a methodical approach. For problems that are more complex, design methods can help to observe a given problem systematically and hence, maximize the amount of considered options for the design. However, many parallel algorithms have already been developed, thus it is not always necessary to start from scratch. Methods that transfer existing parallel algorithms to a certain class of parallel computers, as SMP clusters are very important. Further, another class of methods describe optimization strategies that are often successful, because the structure of many problems is similar. As we can see, methods for the design of parallel algorithms can have several shapes. In the following chapter, we present several methods and make a classification with respect to their position in an overall design process.

Chapter 4

Methods for Designing Parallel Algorithms for SMP Clusters

The design chain for parallel applications, introduced in Chapter 1, is an analytical framework for developing efficient parallel programs. It is possible to analyze and compare different algorithms theoretically and it is guaranteed by the respective programming model that implementations for the target platform behave according to the theoretical prediction. In the last chapter, we presented the κNUMA model, which was the missing element for the design chain for SMP clusters.

In addition to this quantitative basis for the design, we need methods for (1) developing new parallel algorithms from scratch, (2) for transferring parallel algorithms to SMP clusters and (3) for optimizing parallel algorithms with respect to the communication hierarchy of SMP clusters.

With the help of the methods, parallel algorithms can be designed and optimized for SMP clusters. With the κNUMA model it is possible to show the improvement and the efficiency of the resulting algorithms by a theoretical analysis and due to to the corresponding programming model, it is possible to produce efficient implementations. Hence, in order to develop efficient programs, the methods, models and architectures have to be adjusted to each other.

In general, the design of parallel algorithms is a complex task and cannot be reduced to simple recipes. The designer has to deal intensively with the given problem and needs a large portion of intuition, creativity and experience. However, the design profits from a more methodical approach that helps in studying the given problem systematically. According to Foster [31], the aim of such a method should be to maximize considered options, to provide mechanisms to evaluate alternatives and to reduce costs of backtracking from bad choices.

Foster introduced a design methodology that creates a parallel algorithm from a fine-grained problem decomposition. The methodology consists of 4 steps named partitioning, communication, agglomeration and mapping (PCAM). In Section 4.1, we will review this methodology briefly, and will sketch how SPMD algorithms for SMP clusters can be designed using this method.

It is not always necessary to create parallel algorithms from scratch. Many parallel algorithms have been developed for the most computational problems. They were designed under several assumptions like for example a certain cost- or programming model. Hence, the task is to observe these algorithms if they have the capability to perform well on SMP clusters or if they can be changed with respect to the SMP cluster architecture.

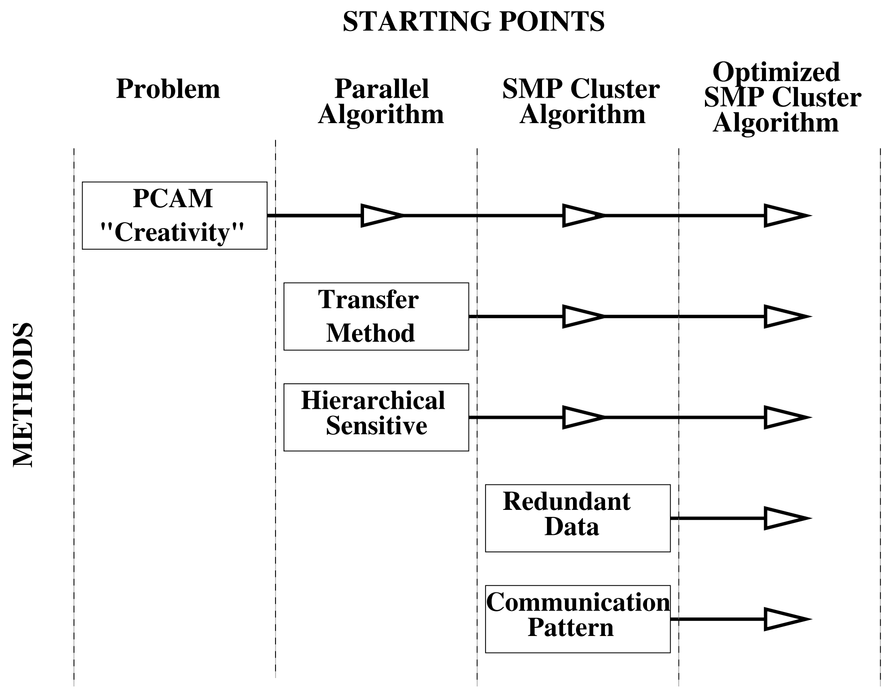


Figure 4.1: Overview on methods for the design of parallel SMP cluster algorithms. The methods have different starting points and may be applied one after the other.

The PCAM methodology is the base for developing parallel algorithms. It builds the frame for the over-all design process. We will show refinements and additional methods that enable the design of efficient parallel algorithms for SMP clusters.

We will present two methods that start on existing parallel algorithms. The first method is a general approach to transfer parallel message-passing algorithms to the hierarchical structure of SMP clusters as described by the κNUMA model. This method is mainly applicable to divide-and-conquer algorithms and is generally explained in Section 4.2. The second method is called *hierarchical sensitive design* and tries to reduce communication costs of a parallel algorithm by dynamically improving the locality of the algorithm during execution, see Section 4.3.

During the transfer of a parallel algorithm to SMP clusters, optimizations that are more detailed are possible. We present two methods that can be applied during the creation of parallel SMP cluster algorithms in order to reduce communication costs and to avoid unnecessary redundant data usage. These methods are generally presented in Section 4.4 and 4.5.

Fig. 4.1 depicts the methods with their respective starting point. In order to create an algorithm for SMP clusters the methods can be applied one after the other. This is not a must, but an opportunity. For certain problems it might not be necessary or even possible to apply all methods and despite of that the resulting algorithm is optimal. Hence, the set of methods can be regarded as a set of guidelines or control points for the overall design process.

4.1 PCAM - Partitioning, Communication, Agglomeration and Mapping

The PCAM method can be applied to a general problem definition. The methodology suggests developing a parallel algorithm for the problem by the following 4 steps.

Partitioning The problem is divided into a number of tasks. Concerning the grain of the tasks, it is suggested to prefer fine-grained tasks, because the method does not want to overlook any possibility for concurrent execution. Either the partitioning can be done by functionality or by data domain. Domain decomposition is the foundation for most parallel algorithms, but functional decomposition is an opportunity to look at the problem in a different way.

Communication The set of tasks, which was created in the last step, is connected by directed edges. Each edge represents dependency from one task to the other. If there is an edge from one task to another, then it is not possible to execute these tasks in parallel, because one task needs the results of the other to perform its computation. After this step, we have already designed a fine-grained algorithm that is represented by a directed acyclic graph whose nodes represent the tasks of the algorithm and the edges represent communication operations.

Agglomeration The preceding steps are architecture independent, because they do not consider architectural parameters that influence performance, like e. g. the number of processors. Now, we review the partitioning into tasks and the resulting communication patterns made in the first steps and take some class of parallel computer into account. In particular, it is observed if it is useful with respect to the target architecture to combine fine-grained tasks in order to provide fewer but coarser tasks. Further, we have to observe, if it is worth replicating data or computation, because sometimes redundant computation is less expensive then getting the result by communication.

Mapping In the final phase, we have to define where each task will be executed. The number of tasks defined by the agglomeration phase may still be greater than the number of processors. Otherwise, the mapping is obvious. The mapping of tasks to processors is a difficult problem. We try to satisfy two conditions that conflict in general. (1) We assign tasks that can be executed in parallel to different processors. (2) We assign tasks that communicate frequently to the same processor in order to increase locality. The mapping problem is known to be NP-complete, hence special strategies and heuristics have to be used.

The agglomeration and mapping steps of the methodology is responsible for optimizing the parallel algorithm that was created by the partitioning and communication phase to a certain class of parallel computer. If we want to adapt the fine-grained algorithm to a SPMD algorithm for SMP clusters, then the agglomeration and mapping phase consists of the following two phases. Like in the κNUMA model, p is the number of processors and N is the number of nodes.

1. We try to create several times p tasks of equal size that do not have data dependencies and hence, can be executed in parallel. Ideally, these tasks can be arranged in levels, where there are always p tasks of equal size per level, and there are only edges of the same direction between the levels.

2. In each level, the p tasks are reduced to N tasks. This second agglomeration is done with respect to minimize the edges and by that communication between the levels.

In the following, we will give a simple example. The problem of making a reduction on n numbers is a basic problem in parallel computation, see e.g. [44]. A set of numbers is reduced to one number by applying a certain operation. In our example, we use the sum of numbers as reduction operation. According to the partitioning phase, we decompose the problem into several smaller tasks. In the most fine-grained decomposition, a task comprises two numbers that have to be added. Hence, this is a functional as well as a domain decomposition.

The communication phase defines the dependencies between the operations. If we regard the tasks as nodes and the dependencies as edges, the resulting graph is a binary tree with height $\log n/2$. In each leaf, one add operation is performed on the initial numbers, and the result is forwarded to their parent node in the tree. Each inner node adds the results of its children and sends the result to its parent node. Finally, the last operation is performed in the root node of the tree and the over-all result is stored there, too (see Fig. 4.2).

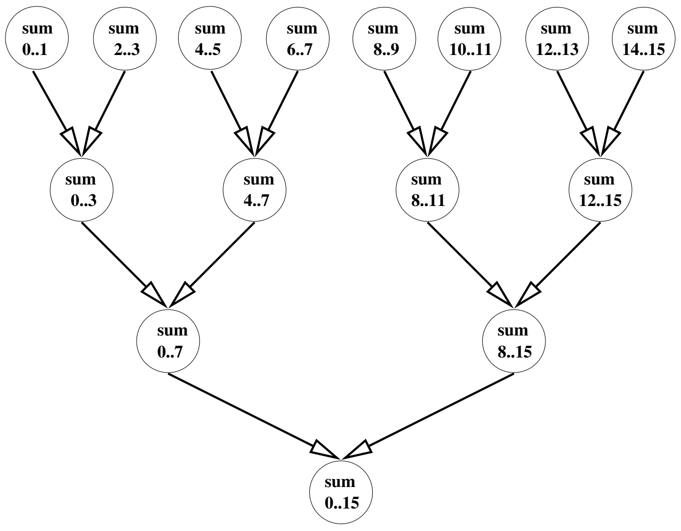


Figure 4.2: Problem decomposition after applying the partitioning and the communication phase to the problem of summing 16 numbers.

If the agglomeration and mapping steps assign the tasks directly to the processors, we have the problem that due to the tree structure only one processor knows the result in the end. Hence, solving the problem this way means to perform an additional broadcast operation of the result. If we want to avoid the broadcast, we could replicate computation and accept more communication operations per level. Hence, in the first phase of the agglomeration step for adapting algorithms to SMP clusters, we use a communication structure called *Butterfly*.

In Fig. 4.3, the resulting algorithm is depicted for 8 processors summing 16 numbers. In each level, each processor performs at least one addition operation and one communication operation. The algorithm terminates after $1 + \log p$ computation and $\log p$ communication steps.

According to the second step of the agglomeration for SMP clusters, we have to reduce the number of task per level from p to N. In Fig. 4.3, the dashed rectangles show the best agglomeration. The resulting graph of the algorithm is illustrated in Fig. 4.4.

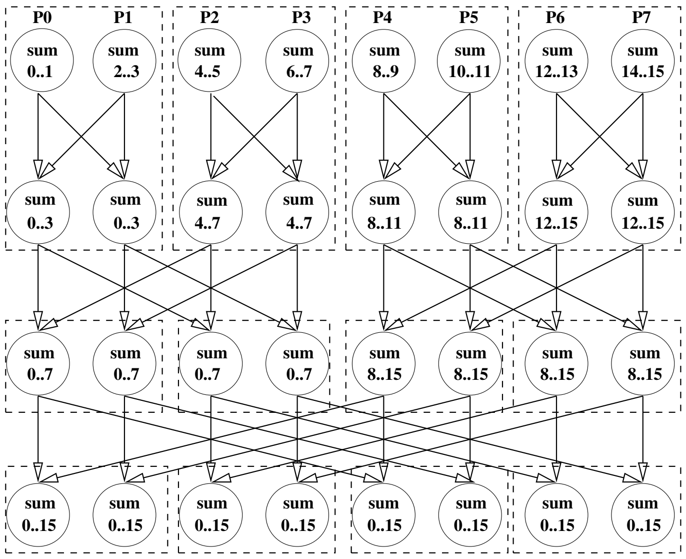


Figure 4.3: Parallel algorithm after the agglomeration and mapping phase for the problem of summing 16 numbers with 8 processors.

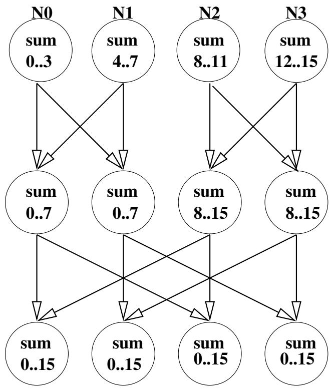


Figure 4.4: Parallel algorithm for summing 16 numbers on an SMP cluster with 4 nodes where each consists of 2 processors. The nodes are denoted with N0, N1, N2 and N3.

The main problem with this methodology is that usually the algorithm designer has a sequential algorithm for the problem in mind. In general, the best parallel algorithm might differ from a parallel algorithm derived from the best sequential one. Hence, there is no guarantee to get the most efficient algorithm, but the method helps in studying the problem deeply and therefore also supports “creativity”. The PCAM method is one possibility to design a parallel algorithm, but it is not a precondition for designing efficient parallel algorithms.

The following methods can be applied to any parallel algorithm, independent on how it was designed and aim to optimize the algorithm to SMP clusters.

4.2 Transferring Parallel Algorithms to Hierarchical SMP Clusters

In SMP clusters, inter-node communication costs are a magnitude higher than accessing local memory. Thus, algorithmic design must attempt to minimize inter-node communication. This is a similar optimization criterion as for designing pure message-passing algorithms. Hence, a successful strategy is to first design an efficient message-passing algorithm, and then adapt the algorithm to the hybrid-programming model. This methodology is a first approach to transfer parallel message-passing algorithms to SMP clusters and is described in [6, 16].

The adaptation is an incremental process, where the computational work that is assigned to each SMP node is mapped into an efficient SMP algorithm. The sequential code is examined for independent operations like IO operations or computational tasks that can be executed by independent threads. The most successful case for multi-threaded execution is the concurrent execution of loop-nests. Each thread executes its part of the iterations independently. Loop transformations may be necessary to reduce the data dependencies between the threads and thus to increase parallelism. In [41], two sorting algorithms are presented that were transferred to SMP clusters using this method.

As we know from Section 2.4.2, the result of this strategy in contrast of using a pure message-passing based algorithm depends on several parameters. The most success can be achieved if the algorithm belongs to the class of divide-and-conquer algorithms and there are several levels of hierarchy between the computing nodes. Therefore, we want to reformulate the method for the κNUMA model and for divide-and-conquer algorithms.

It seems to be easy to apply message-passing algorithms to SMP clusters because, ultimately, the system is also only a set of processors, which are connected by a network or shared-memory. However, this assumption is not optimal, because of the differences between inner- and inter-node communication. This difference even increases for larger hierarchies. The communication hierarchy enlarges the internal memory hierarchy of SMP nodes and has great influence on the performance of parallel algorithms.

Thus, algorithms should be developed in a way that communication takes place inside the SMP nodes or in the lowest level possible of the network. In contrast to κNUMA, the other models have the same costs for all inter-processor communication, independent of where they are situated.

In order to achieve the best running time on κNUMA, it is obvious to minimize the communication in each level of the machine and therefore the total communication. We suggest a top-down approach, because the higher the level of hierarchy in the network, over which we have to communicate, the more expensive communication will be. First, we have to develop an efficient message-passing algorithm for the virtual machine with $\alpha(\kappa)$ $(\kappa-1)$NUMA-sub-machines. This algorithm consists of several partial problems, which all have to be solved by efficient message-passing algorithm for the virtual machine of the next level (with $\alpha(\kappa-1)$ $(\kappa-2)$NUMA-sub-machines). This procedure has to be continued until the level of the real SMP nodes is reached. The problems on the SMP nodes have to be solved by an efficient shared-memory algorithm. An efficient SMP-algorithm consists of parts that will be executed sequentially on each processor and on synchronized shared-memory operations. Only if we use efficient sequential code and data structures we will have optimal SMP-algorithms. Using this method guarantees an efficient transfer from non-hierarchic systems to κNUMA machines, hence, it is called κ*NUMA method*.

The design of an optimal algorithm and its analysis gets easier if it is possible to use the same message-passing algorithm for all κNUMA-sub-machines. This is possible, as soon as the algorithm divides the input problem into a smaller instance of the same problem (divide-and-conquer).

As an example for this approach, we review the problem of summing n numbers again. Obviously, this problem can be solved by a divide-and-conquer algorithm. The starting point for the method is the parallel algorithm explained in the previous section, see Fig. 4.3. The complete algorithm can be constructed by recursively applying this algorithm in a top-down fashion to all levels of the κNUMA machine. That means that we first apply the algorithm to the $\alpha(\kappa)$ sub-machines of level κ. These sub-machines have to sum $n/\alpha(\kappa)$ numbers and use their $\alpha(\kappa-1)$ sub-machines to solve this problem (call of recursion). The end of the recursion is reached if the algorithm can be applied to the SMP nodes. Each SMP node has to build the sum of n/N numbers with its $\alpha(0)$ processors.

Basically, we get the same SMP cluster algorithm as with the PCAM method. However, the methods are totally different. While PCAM suggests a fine-grained bottom-up strategy starting with the problem partitioning, the κNUMA-method uses a recursive top-down approach applying well-proven parallel message-passing algorithms to each level.

We have already presented another example for the method in Chapter 3. The κ-level algorithm was constructed by applying the direct broadcast algorithm from the top level of the hierarchy to the SMP nodes. The analysis showed that this version is superior to the original algorithm.

4.3 Hierarchical Sensitive Design

What we learned from the κNUMA model is that in hierarchical SMP clusters, algorithms should attempt to split their work into more and more independent pieces in order to reach higher locality during the execution. For non-hierarchic distributed-memory systems, it does not matter which processors have to communicate, because the costs are the same. In hierarchic systems, we should attempt to minimize the communication operations for each level in the network.

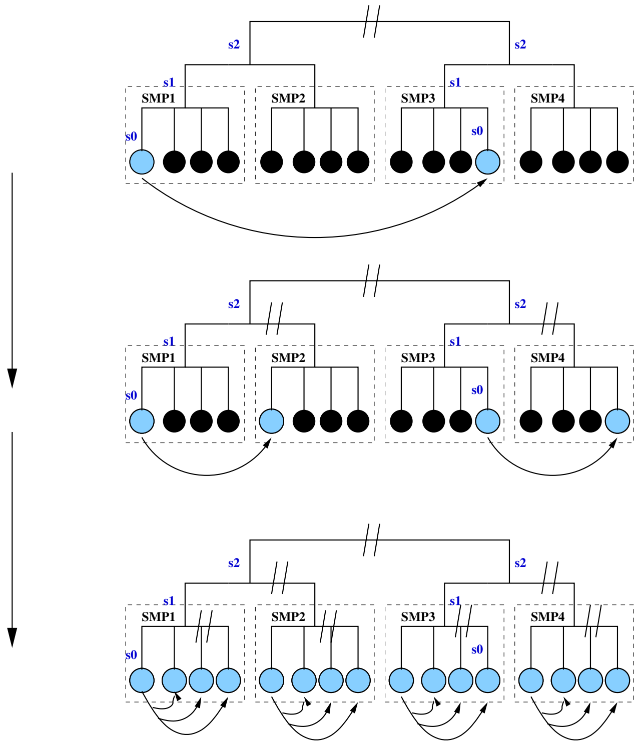


Figure 4.5: Hierarchic sensitive broadcast algorithm for hierarchic SMP clusters. The algorithm needs three communication steps to perform the broadcast. The level of hierarchy involved in the communication decreases with every step.

An algorithm can be called *hierarchical sensitive*, if it minimizes the communication operations per level and prefers local computation instead. In consecutive communication steps, only levels of equal or lower height should be used. Normally, the algorithm terminates with a final local computation.

As an example, we revisit the broadcast problem for a κNUMA machine. In the first step, the message is transferred over the highest level of the network. This splits the task into several tasks ($\alpha(\kappa)$ tasks) that can be solved by the receiving processors concurrently. For all following communication steps, it is guaranteed that the height of the actual level is lower than the height of the level of the previous communication step. Finally, the algorithm terminates by a local memory copy of the message, compare Fig. 4.5.

The broadcast example is straightforward for the method, because the problem can be easily partitioned into several independent tasks. In general, such decomposition is not achieved easily for the general case. Generally, the method can be applied to problems that can be solved by the iteration of two steps. First, informations about local data are exchanged between groups of processors. By exchanging these informations, a global knowledge of the distributed data is created. Second, due to the global knowledge the data can be redistributed. Further iterations of the two steps will involve smaller and smaller groups of processors. With decreasing group sizes, the level used for the communication operations will get lower and lower. This is repeated until only local computation remains.

4.4 Adaptation of Communication Patterns

In non-hierarchic distributed-memory machines, a certain communication pattern leads to the same costs independent of which processors are involved in the operation. As we saw by the analysis of the broadcast problem on the κNUMA model, in hierarchic systems, this has great influence on the communication costs.

Looking at the broadcast algorithm again, despite of using the same communication pattern, it is possible to deteriorate the running time by not adapting the pattern efficiently to the architecture. In Fig. 4.6, we use the same pattern as in the example presented in Fig. 3.1. The difference is the second communication step that is done inefficiently, because the two communication operations have to cross the highest level in the network again which is unnecessary.

In the example, the communication pattern is static; the only thing we can change is the order in which the processors are considered during the execution. By choosing a distinct order, we get a more or less efficient instance of the broadcast algorithm. Hence, the example is very limited concerning the influence on the communication pattern. Generally, in SPMD algorithms, a processor makes computations on local data and after that a certain communication pattern is executed. Normally this pattern depends on the respective data distribution that was chosen for the computation. Hence, the task is to find a data-distribution that influences the next communication pattern in the way that it is maximally adapted to the underlying SMP cluster architecture.

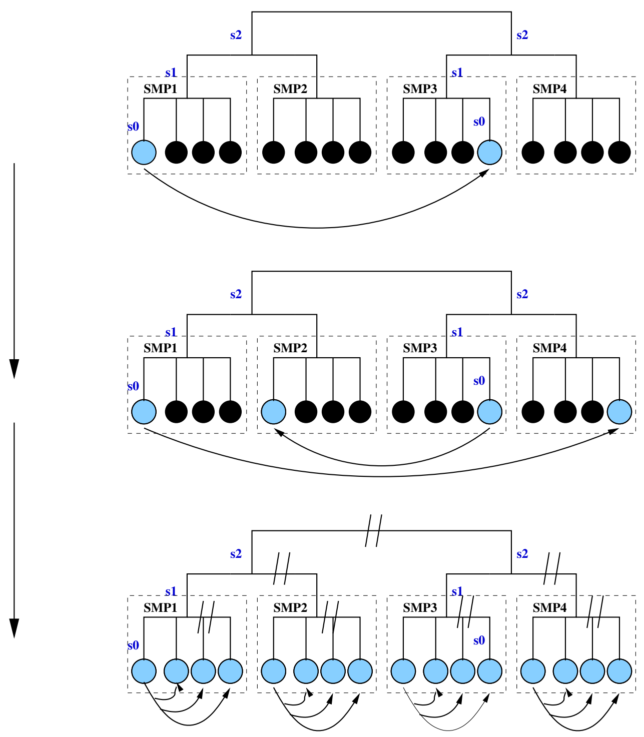


Figure 4.6: Broadcast algorithm for a hierarchic SMP clusters with a non-adapted communication pattern. The algorithm needs three communication steps to perform the broadcast. The level of hierarchy involved in the communication does not decreases with every step.

4.5 Usage and Avoidance of Redundant Data

In general, an algorithm should avoid storing data redundantly. On the other side, an algorithm can also profit from redundant data. For example, an algorithm has the possibility to make computations on these data that normally are done by other processors, too. Instead of receiving the results by communication, it can compute the results locally. Whether such a strategy is successful depends on the ratio between the extra computation and the saved communication costs.

A more attractive situation is reached, if it is possible to use redundant data in the way that the number of local computations stays the same, but the structure of the algorithm leads to a reduction of communication operations in further steps. This may arise, if due to the redundant data final, or at least less partitioned intermediate results can be computed.

For example, the algorithm for the parallel reduction of n numbers presented in Fig. 4.3 stores intermediate results of the sum redundantly in each processor. Due to this investment, it is possible for each node to compute the final result step by step and by that an expensive broadcast operation of the final result is avoided. The additional computation each processor has to do can be hidden by parallelism. Although we save a complete broadcast operation and do not need any additional computation that increases the running time, the situation is not optimal. We have more point-to-point communication operations per iteration in order to make the redundant intermediate results accessible to the processors. Despite of that the example shows the potential of the method.

In general, optimal situations exist for algorithms that work on multi-dimensional data grids. Usually, the grid is distributed evenly among the processors. Each processor stores a multi-dimensional block of the whole grid of the same size. The shape of this part can be changed by increasing and decreasing its dimensions. The amount of computation is normally directly dependent on the size of this local block. Hence, the amount of computation per processor can be preserved, if the increase of one dimension is compensated by the decrease of another dimension. The compensation does not lead to redundant data, but in general, more than one multi-dimensional grid is involved in the computation. The shape and size of these additional blocks depends on the shape of the main block and the operation that is performed. Hence, redundant data is very likely concerning the additional blocks. In general, such operations produce intermediate results that have to be reduced to the final result by at least one more communication and computation step. Hence, the initial data distribution has effect on the preceding communication step, because it defines which processor computes which intermediate results and thus defines the communication pattern. An investment in redundant data may result in a more efficient communication operation without increasing computation time.

Another aspect of the method is that we want to avoid unnecessary usage of redundant data. In general, data is distributed evenly among the processors in order to reach a good load balance. For non-hierarchic machines this is done in an anonymous fashion, because it does not matter to which processor the data is assigned. In contrast to that, if we look at κNUMA machines the assignment has great importance. Basically, the whole data for one processor consists on several structures. The aggregation of these structures over all processors is normally not a disjoint set of the whole data, because there is data that is used by all or a sub-set of all processors. Hence, in order to reduce the over-all amount of data in each SMP node, we should assign jobs that need mostly the same data to processors of the same SMP node. By assigning the data this way, it is possible to store the data structures that are needed for multiple processors only once per SMP node by using an appropriate programming model like e.g. the hybrid-programming model that uses threads within the nodes that have a common memory space. Despite an appropriate programming model, without the grouping of the processors according to the data they use, such a minimization of the amount of memory is not possible. Therefore, the usage of redundant data on SMP clusters is not as expensive as for distributed-memory systems where each processor has its own memory space. If we assume that an amount of data has to be accessed by all processors then we only have to store it once per SMP node. Depending on the number of processors per node $\alpha(0)$, this might even become insignificant. Hence, algorithms that use redundant data profit from the SMP cluster structure. The total amount of memory per node can be reduced which is especially important for large-scale applications.

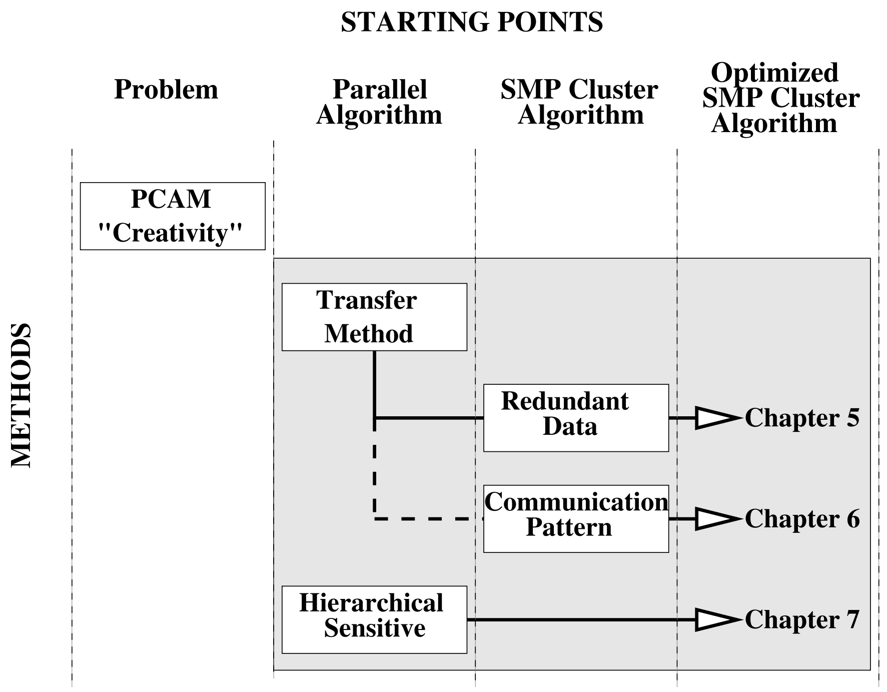


Figure 4.7: Overview on methods for the design of parallel SMP cluster algorithms.

4.6 Summary

In the last sub-sections, we introduced and reviewed several methods for designing parallel algorithms and for optimizing them with respect to hierarchical SMP clusters. The general design process for a parallel algorithm for SMP clusters has several entry or starting points for the application of the methods. While some methods assume nothing else than the problem definition, others start on well-proven parallel algorithms.

Fig. 4.7 summarizes the different starting points of the methods. The gray shaded region illustrates the scope of the remaining part of the thesis.

All these methods are applicable on parallel algorithms in order to create efficient parallel algorithms for SMP clusters. The usage of these methods is therefore independent on the way these parallel algorithms were developed. The designer is free to use the methodical approach PCAM, his "creativity" or if possible already existing parallel algorithms. While in the last sections we explained the methods using simple examples and general explanations, in the following, we will show the use of the methods by detailed case studies for more complex problems. The verification of the efficiency of the resulting algorithms is done by analyses on base of the κNUMA model where necessary. The theoretical predicted results are verified practically by experimental tests. The individual path of development for these case studies is visualized by the directed edges in Fig. 4.7.

Chapter 5

Exploitation of Data Redundancy

One observation made from looking at the SMP cluster architecture is that data can be used redundantly by multiple processors within the SMP nodes. Using the parallel dense matrix-vector-multiplication as an example, we show the possibility to reduce communication cost of algorithms by storing data redundantly but without increasing the per processor amount of memory in an unacceptable way. On the other side, sometimes the minimization of memory is more necessary than reducing communication costs. For this case, we show that the use of the κNUMA model together with the appropriate hybrid-programming model avoids the storage of unnecessary redundant data.

5.1 Adapting κNUMA to the Target Platform

Parameter	Description
$N = \alpha(1)$	Number of SMP-nodes
$\alpha(0)$	Number of processors per node
s_0	Access time to local memory
s_1	Access time to the network
g_{global}	Bandwidth of the network
g_{local}	Bandwidth within SMP-node

Table 5.1: Reduced set of parameters for a 1NUMA machine

Since we want to make the theoretical analysis more feasible, we adapt the κNUMA model to our target platform, called *Kepler-Cluster*. The Kepler-Cluster [75] of the University of Tübingen is a Linux-based cluster, which consists of 98 computing nodes based on dual BX-Boards with Pentium III at 650 MHz clock rate[1]. Each board has 1 GB of memory and is connected both to a Myrinet LAN and to an Ethernet network. The Kepler-Cluster was included three times in the *TOP500* [60] list (November 2000 until November 2001). Although the interconnection network has an hierarchical structure, it is not possible to measure different message-passing times for the different levels, because the overhead for a message is much higher (μs) than the additional times for each level (ns). Hence, our target machine corresponds to a 1NUMA-machine that reduces the set of parameters as illustrated in Table 5.1.

5.2 Analysis of the Parallel Dense Matrix-Vector-Multiplication in Distributed-Memory Systems

In this section, we analyze the problem of multiplying an $n \times m$ matrix $\mathcal{A}$ with an m-element vector x in parallel, where both are dense. The resulting vector b consists of n elements ($\mathcal{A}x = b$). The straightforward sequential algorithm for this problem is a nested loop that iterates over the matrix and the vector in the following way[2].

```
for (int i=0; i<n; i++) {
    for (int j=0; j<m; j++) {
        b[i]+=A[i][j]*x[j];
    }
}
```

As we can see, the running time can be asymptotically charged as $O(nm)$. Optimizations of this code according to the cache utilization are possible and depend on the actual memory layout of the matrix and the vector and the kinds of caches and their sizes for the respective platform. Generally, this approach assumes that the matrix as well as the vector is stored in the same memory space. Under this assumption, it is very simple to make a parallel shared-memory version. As we know from Chapter 2.4.1, OpenMP can parallelize loop nests by defining parallel regions and work-sharing directives. The code above is extended by a parallel *for* directive, but the nested loop itself stays unchanged. As a result the iterations are divided equally among the threads and therefore among the available processors. By default, the variables are shared between the threads. The schedule directive has the argument *static* that defines an equal distribution before runtime. In this case, this is the best schedule strategy, because the work

[1]Currently, the Kepler-Cluster is extended by 32 additional nodes.

[2]We use C code fragments to illustrate the algorithm and assume that the variables (`i,j,n,m,b,x,A`) are declared in the complete program.

that has to be done is deterministic and thus, no synchronization for redistribution during the loop is necessary. More information about the clauses of the directives can be found in the OpenMP specification [61].

```
#pragma omp parallel for schedule(static)
for (int i=0; i<n; i++) {
    for (int j=0; j<m; j++) {
        b[i]+=A[i][j]*x[j];
    }
}
```

In contrast, the situation in distributed-memory systems is more complex. If we want to speed-up the computation it is of course also possible to distribute the iterations equally, but the problem is that we do not have access to a shared result vector. That means that each processor can only produce partial results, which have to be combined later. Further, in the shared-memory version the whole data is stored only once for all processors. We want to achieve the same situation for the distributed case, because it is inefficient to store the whole data for each processor separately.

5.2.1 Basic Parallel Algorithm

Roughly speaking, each processor stores a block of equal size from the matrix $\mathcal{A}$ in its local memory. Additionally, each processor stores the part of the vector x which is necessary to perform a partial multiplication with its local block of the matrix. For simplicity, we neglect the arithmetic precision and assume that the data can be partitioned evenly among the processors.

To be more precise, the data-decomposition can be described by two parameters. Let h be the number of blocks in horizontal direction and v the number of blocks in the vertical direction of the matrix $\mathcal{A}$. It follows directly that $p = hv$. Therefore, each processor stores $\frac{nm}{p}$ elements of the matrix $\mathcal{A}$ and $\frac{m}{h}$ elements of the vector x.

Hence, the resulting algorithm consists of three phases namely, local matrix-vector multiplication, communication of partial results and accumulation of the received partial results, see *Algorithm 1*.

After the execution of *Algorithm 1*, the result vector b is distributed equally among the processors. Since we do not know the context in which the multiplication takes place, we do not consider the collection or storage of the result vector for the algorithm. Variants of this algorithm for different computational models can be found in standard textbooks. See for example [44] and [54].

Algorithm 1: Basic parallel dense matrix-vector-multiplication algorithm

1. Phase: Each processor computes a partial result vector by multiplying the local block of the matrix $\mathcal{A}$ with its part of vector x of size $\frac{m}{h}$. The resulting vector has the length $\frac{n}{h}$.
2. Phase: Each processor sends parts of the partial result vector to the corresponding $v-1$ processors. These parts have the length $\frac{n}{p}$.
3. Phase: Each processor adds all received partial vectors of size $\frac{n}{p}$ to its own partial vector. After that, each processor has $\frac{n}{p}$ continuous elements of the result vector b.

Optimization Directions

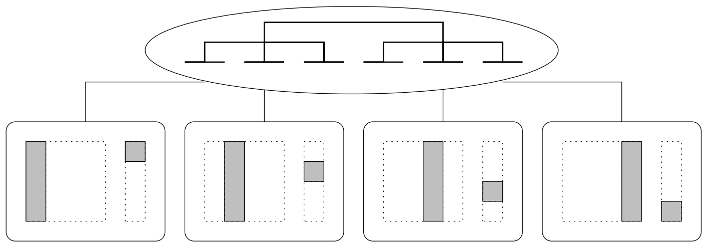
Looking at the algorithm, we notice that the initial distribution of the matrix over the processors defines the number of necessary communication operations and determines the length of the vector, which has to be stored in each processor. In order to minimize the number of communication, we have to set v as low as possible and h as high as possible. If we want to minimize the initial memory space and want to avoid redundant storage of vector elements, then we have to set v and h vice versa. Hence, the matrix decomposition is the major factor in determining the efficiency of the basic algorithm. In the following analysis, we want to compare three different situations. First, the minimization of initial memory usage ($h = p$, $v = 1$), second, minimization of the number of communication operations ($h = 1$, $v = p$) and, third, the balanced situation ($h = v$, if p is square).

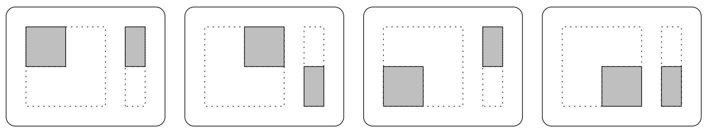
5.2.2 Data Distribution

The three considered distributions are illustrated in Fig. 5.1. Each node stores the part of the vector, which is necessary for the multiplication with its part of the matrix, in the local memory too. Again, the input matrix $\mathcal{A}$ has size $n \times m$.

1. **Vertical stripes (vstripe)**, $h = p$, $v = 1$.
   Each node works on a matrix with size $n \times \frac{m}{p}$ and on a vector with size $\frac{m}{p}$ (see Fig. 5.1a).
2. **Square blocks (block)**, $h = v = \sqrt{p}$.
   Each node works on a matrix with size $\frac{n}{\sqrt{p}} \times \frac{m}{\sqrt{p}}$ and on a vector with size $\frac{m}{\sqrt{p}}$ (see Fig. 5.1b).

3. **Horizontal stripes (hstripe),** $h = 1, v = p$.
   Each node works on a matrix with size $\frac{n}{p} \times m$ and on a vector with size m (see Fig. 5.1c).


(a) vstripe


(b) block

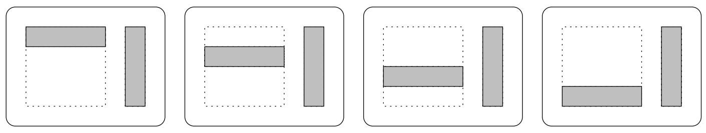

(c) hstripe

Figure 5.1: Different initial data decomposition strategies for the matrix and the vector. The shaded regions have to be stored in the nodes ($N = 4$).

Corresponding Algorithms

Now we have to adapt the basic algorithm to the distributions *vstripe*, *block* and *hstripe*. This can be done by setting the values of h and v of the distributions into the basic algorithm. The resulting numbers are presented in Table 5.2. As an example, the adapted algorithm for *hstripe* is shown in Fig. 5.2.

Transfer of the Basic Algorithm to the Parallel Hierarchy

Until now, we have described a basic algorithm for the parallel dense matrix-vector multiplication, we have pointed out that depending on the data decomposition we can optimize either the number of communication operations or the initial memory space. However, we have not considered the parallel hierarchy of the target platform. Applying the algorithms directly

Distribution	Communications	Initial Memory Space
vstripe	p	$\frac{nm}{p} + \frac{m}{p}$
block	$\sqrt{p}$	$\frac{nm}{p} + \frac{m}{\sqrt{p}}$
hstripe	0	$\frac{nm}{p} + m$

Table 5.2: Number of communication operations and the size of the initial memory space of the basic algorithm for the considered distributions

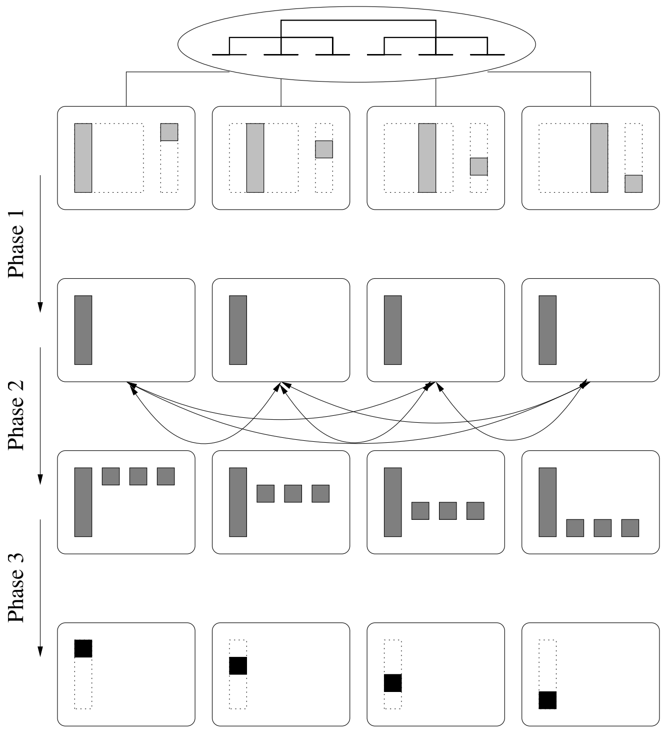


Figure 5.2: Three phases of the parallel matrix-vector-multiplication algorithm for the *hstripe* decomposition. ($N = 4$).

to the target architecture without considering the hierarchical structure can lead to inefficient algorithms. We will give an example in Section 5.2.4.

The presented algorithm is a *divide and conquer* algorithm that solves the problem by solving the same problem on smaller instances of the input. In the 1. phase of the basic algorithm, a matrix-vector multiplication on the locally stored data is called. Concerning the hierarchical structure, we can apply the basic algorithm from the top level to the lowest level of the κNUMA machine. For each level, it is possible to use another data distribution dependent on the characteristics of the respective sub-machine. Our target machine is a 1NUMA machine that has only two levels, the cluster level and the node level. The main difference is the memory system. There is distributed-memory between the nodes and shared-memory within the nodes. Hence, the question, we have to answer is, which data decomposition for each level is the best. This situation is illustrated in Fig. 5.3.

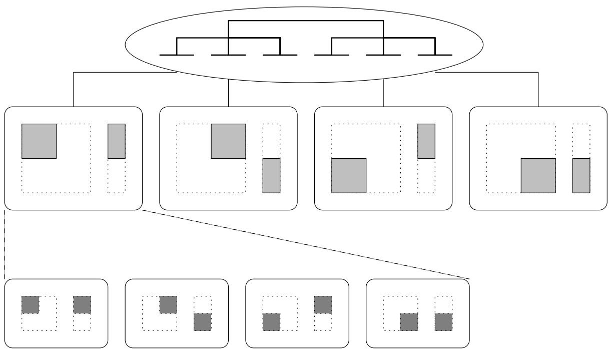


Figure 5.3: Parallel hierarchy in the SMP cluster

5.2.3 Analysis

In order to create an overall algorithm, we have to stick the algorithms for all levels together. Due to our target platform, we have a cluster-level algorithm ($\kappa = 1$) and a node-level algorithm ($\kappa = 0$). The three phases of both algorithms are denoted *X*Phase*Y*, whereby *X* is either C for cluster-level or N for node-level and *Y* stands for the number of the concerned phase. The overall algorithm is illustrated in Fig. 5.4. The shaded regions are obsolete, if the *hstripe* decomposition is used either at the node- or cluster-level, because in this case each processor computes directly values of the result vector and hence, no further communication or computation is necessary.

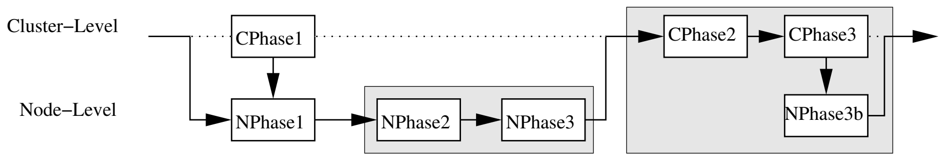


Figure 5.4: Phases of the overall algorithm.

Cluster Algorithms Phase 1

Table 5.3 gives an overview of asymptotic computation and communication times for the three considered distributions for CPhase1. CPhase1 consists of NPhase1, NPhase2 and NPhase3 (see Fig. 5.4). While NPhase1 is equal for all distributions, due to NPhase2 and 3 we get a unique ranking: 1) hstripe, 2) block, 3) vstripe.

Distribution	NPhase1	NPhase2	NPhase3
vstripe	$O\left(\frac{nm}{N\alpha(0)}\right)$	$2s_0 + \frac{2n}{g_{local}}$	$O(n)$
block	$O\left(\frac{nm}{N\alpha(0)}\right)$	$2s_0 + \frac{\frac{2n}{\sqrt{N}}}{g_{local}}$	$O\left(\frac{n}{\sqrt{N}}\right)$
hstripe	$O\left(\frac{nm}{N\alpha(0)}\right)$	0	0

Table 5.3: Analysis of node-level algorithm for phase 1 of the cluster-level algorithm (CPhase1)

Cluster Algorithms Phase 2

The main thread of the node process is responsible to do the communication operations of phase 2. Depending on the distribution, each node has to send $N-1$, $\sqrt{N}-1$ or 0 messages. Table 5.4 shows the three different formulas for the distributions. Concerning CPhase2, the ranking is 1) hstripe, 2) block, 3) vstripe again.

Distribution	Communication
vstripe	$Ns_0 + 2s_1 + \frac{n-\frac{n}{N}}{g_{global}}$
block	$\sqrt{N}s_0 + 2s_1 + \frac{\frac{n}{\sqrt{N}}-\frac{n}{N}}{g_{global}}$
hstripe	0

Table 5.4: Analysis of phase 2 of the cluster-level algorithm for the different distributions (CPhase2)

Cluster Algorithms Phase 3

In the last phase of the cluster-algorithm, each node adds its received partial result vectors in order to compute a part of the result vector of size $\frac{n}{N\alpha(0)} = \frac{n}{p}$ (Table 5.5). Again the algorithm behaves different for the three distributions, but the ranking stays the same: 1) hstripe, 2) block, 3) vstripe.

Ranking

Summarizing the results of Table 5.3, 5.4 and 5.5, the ranking of the data decompositions is obvious. It is not necessary to aggregate the contents of all tables for computation and communication, because the ranking of all data decompositions in all phases of the cluster-algorithm is identical. Looking at the node-level algorithm (CPhase1), the ranking is clearly *hstripe*, *block* and then *vstripe*. Within the nodes, we do not have the problem of using

Distribution	Computation
vstripe	$O\left(\frac{n}{\alpha(0)}\right)$
block	$O\left(\frac{n}{\sqrt{N}\alpha(0)}\right)$
hstripe	0

Table 5.5: Analysis of phase 3 of the cluster-level algorithm for the different distributions (CPhase3)

too much memory for *hstripe*, because there is shared-memory. The whole vector for the partial problem has to be stored in the node anyway. Hence, the ranking for the node-level algorithm stays unchanged. Looking at the cluster-level algorithm, the decomposition with the best resulting performance is *hstripe*, the second best is *block* and the third best is *vstripe*. Of course, this is the ranking in order to optimize performance. If it is necessary to use as little memory as possible, then the ranking is vice versa for the cluster-level algorithm. All SMP nodes have to store the same amount of matrix elements for all distributions. This is not true for the vector elements. Using hstripe means to store the whole vector in each node (m elements). If block-distribution is used, then only $m/\sqrt{N}$ elements have to be stored in each node. The least vector elements per node have to be stored when the vstripe decomposition is used. Here it is only necessary to store m/N elements of the vector, which is optimal. Nevertheless, because of the parallel hierarchy the situation is not as bad as it would be for BSP-machines. Table 5.6 shows the initial memory usage per processor for the three variants of the cluster-level algorithm. It is not necessary to store the whole vector for each processor, we only have to store it in each node. This makes *hstripe* more efficient, especially if $\alpha(0)$ is not much smaller than $N\alpha(0)$.

Distribution	Initial Memory Space
vstripe	$\frac{nm}{N\alpha(0)} + \frac{m}{N\alpha(0)}$
block	$\frac{nm}{N\alpha(0)} + \frac{m}{\sqrt{N}\alpha(0)}$
hstripe	$\frac{nm}{N\alpha(0)} + \frac{m}{\alpha(0)}$

Table 5.6: Initial memory usage per processor of the overall algorithms.

5.2.4 Problems Involved with a Non-Hierarchical Approach

Before we show results of experimental test for previous analysis, we want to look back briefly to non-hierarchical analyses. In the following, we will

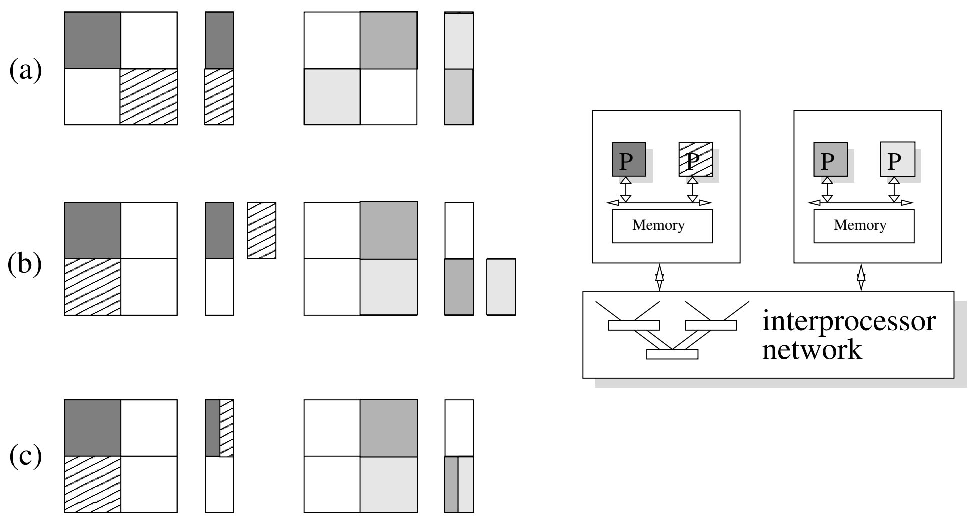


Figure 5.5: Problems with a non-hierarchical approach may appear using the *block* distribution concerning redundant data. Depending on the assignment of blocks to processors, vector elements have to be stored more than necessary in each node. The quadrats represent the matrix and the vertical rectangles represent the vector. The assignment of blocks to processors is illustrated by the shading.

show that implementations based on these analyses may lead to inefficient memory usages. In [54] the *block* distribution is used for the matrix-vector-multiplication. As we explained before, this leads to a compromise between communication cost and memory usage. However, this is not the whole story. In Fig. 5.5, three situations are presented that behave differently concerning the amount of memory per SMP node, although they all belong to the *block* distribution. The difference is in the mapping of the blocks to the processors. As an example we take a SMP cluster that consist of two nodes and each node has two processors. According to the non-hierarchic approach, it is not important which part of the matrix and vector is assigned to which processor. Therefore, in Fig. 5.5a an assignment is shown which leads to the fact that the whole vector has to be stored for two processors. According to non-hierarchical models this is optimal. However, if we do the assignment as we show in Fig. 5.5b, then the processors in each node can use the same part of the vector. If we use processes for the computation of the partial results, then we have the same problem as in the Fig. 5.5a, because each process has its own memory space and therefore, the part of the vector is stored redundantly although it is identical. But in the κNUMA model and the corresponding hybrid-programming model, threads of the same node have a common memory space. Hence, the identical part of the vector for both computations can be stored only once, compare Fig. 5.5c. This problem will not appear if the analysis respects the hierarchical architecture as we showed in the previous sections. Looking at the example, we can realize that Fig. 5.5c is nothing different than choosing the *vstripe* algorithm for the cluster-level and applying the *hstripe* algorithm for the node-level algorithm. As we know from the analysis this should be the variant with the lowest memory requirements and with the fastest node level algorithm. Indeed the cluster-level algorithm needs communication and is therefore not as fastest choice. In the next section, we will show results of experimental test that will confirm the ranking from the theoretical analysis of the previous sections.

5.3 Experimental Tests

The experimental tests for the cluster-level algorithm were made on different instances of the Kepler-Cluster [75] of the University of Tübingen. The experiments for the SMP-algorithms were made on one node of this cluster and on a Sun Sparc Ultra-4 Workstation with 4 processors. The program was written in C++ using Posix-Threads for SMP programming and TPO++[38] (Section 2.4.1) for message-passing. For all experimental tests we set $m = n$.

We decided not to use OpenMP, because we wanted to program the different data distributions for the SMP node algorithm explicitly in order to make the results comparable. To rely on the scheduling possibilities of OpenMP would have been more convenient, but it is more difficult to formulate the three distributions.

SMP-Node Experiments

All experimental tests on one SMP node confirmed the theoretical ranking of the three data decompositions. In Table 5.7, we illustrate an example for the Sun workstation ($\alpha(0) = 4$) and in Table 5.8, there is an example for one node of the Kepler-Cluster ($\alpha(0) = 2$).

n	Distribution	time in sec
4096	vstripe	3.31
	block	3.17
	hstripe	2.74
2800	vstripe	1.62
	block	1.48
	hstripe	1.29

Table 5.7: Tests for the SMP-algorithm on the Sun Sparc Ultra-4 (4 CPUs).

n	Distribution	time in sec
4096	vstripe	2.62
	hstripe	2.52
2800	vstripe	1.3
	hstripe	1.18

Table 5.8: Tests for the SMP-algorithm on one node of Kepler (two CPUs). The block distribution is not possible for two CPUs per node.

Cluster Experiments

The SMP experiments confirmed the predicted ranking of the distributions. Now we want to confirm the predicted ranking of the distributions concerning CPhase2. It does not matter which distribution we choose for the node-level algorithm. In the experiments, we take the hstripe distribution, because it is the fastest. In Table 5.9 and 5.10, we can see that the experiments confirm the predicted ranking for the communication phase (CPhase2).

n	Distribution	CPhase1	CPhase2	CPhase3
11200	vstripe	1.48	0.04	0.01
	block	1.38	0.01	0
	hstripe	1.41	0	0
5600	vstripe	0.46	0.02	0.01
	block	0.44	0.01	0
	hstripe	0.35	0	0

Table 5.9: Tests on the Kepler-Cluster with 16 nodes (time in sec). Values that are 0 stand for times < 0.01.

However, it is also obvious that the time for CPhase2 is so small that it is nearly not relevant for the total runtime of the algorithm. The runtime of the algorithm is dominated by the runtime of the node-level algorithm (CPhase1). By using more nodes, it is possible to reduce the overall runtime by an optimal speed up. This works well, because the algorithm has a very good scalability. Nevertheless, the more nodes are involved, the more communication operations are necessary. With an increasing number of nodes, the communication part of the runtime becomes more and more important and, therefore, the programmer should consider the ranking of the three distributions for his choice. Although we use the same algorithm (hstripe) for CPhase1 in the cluster experiments and the amount of data involved in the computation is equal, the timings are different for the three distributions. These differences are likely depending on cache utilization, because the arrays have a different shape depending on the distribution used in the cluster-level algorithm. The CPhase3 can be computed very fast for all distributions. It does nearly not contribute to the overall runtime.

Nodes	Distribution	CPhase1	CPhase2	CPhase3
16	vstripe	4.73	0.08	0.01
	block	4.29	0.02	0
	hstripe	4.68	0	0
25	vstripe	2.95	0.09	0.01
	block	2.75	0.01	0.01
	hstripe	3.03	0	0
49	vstripe	1.59	0.10	0
	block	1.43	0.01	0
	hstripe	1.54	0.01	0

Table 5.10: Tests on the Kepler-Cluster with 16, 25 and 49 nodes where n=19600 (time in sec). Values that are 0 stand for times < 0.01.

5.4 Summary

In this chapter, we illustrated the use of the κNUMA-method that enables the transfer of algorithms from more general message-passing models like BSP. The main idea is to use efficient algorithms for each level of the machine in a top-down fashion. This recursion ends at the SMP node level where efficient shared-memory algorithms are used for the computation.

As an example, we developed efficient SMP cluster algorithms for the parallel dense matrix-vector multiplication. In contrast to the analysis of broadcast problems in Chapter 3, we made the κNUMA model more feasible by adapting the parameters to a certain SMP cluster, and we made experimental tests.

Besides applying the κNUMA method, we observed the algorithm with respect to the usage of redundant data. We showed that the data decomposition plays a key-role for the efficiency of the algorithm. By analyzing the different variants of the algorithms, we predicted a ranking for the experimental results on a Linux cluster with two processors per node. The experimental tests confirmed the predicted ranking of the analyzed algorithms. Therefore, the κNUMA framework is suited for the development of algorithms for SMP clusters.

We showed that within the nodes the *hstripe* distribution is the best choice for the algorithm, because of the shared-memory between the processors. It is also the best choice for the cluster-level algorithm, if there is enough memory to store the whole vector x in each node. An important conclusion of the analysis and the experimental tests is that it is possible to improve the performance by using redundant data. By storing more elements of vector x in each node, it was possible to use less communication. This is a technique, which might as well be useful for other problems.

In addition, the investment in redundant data is not as high as it would be predicted by the BSP model. It is only necessary to store $m/\alpha(0)$ elements of vector x per processor. The optimum is $m/A\alpha(0)$ what means that in some architectures, where the number of nodes A is small, the difference is not relevant at all.

Further, we showed an example that the κNUMA method avoids an unintentional redundant storage of data. Due to the hybrid-programming model it is possible to distribute data in a way that those processors who reside in the same node mainly have to work on the same data. This technique saves memory space, which is a critical resource in large scale applications.

Chapter 6

Adaptation of Communication Patterns

In general, parallel algorithms contain communication steps where processors communicate according to a certain pattern. In Chapter 3, the personalized broadcast was introduced which is an example for a one-to-all pattern. In Chapter 5, where the parallel matrix-vector multiplication was analyzed , the all-to-all communication pattern was introduced, where each processor sends messages to all other processors. We have already seen at the analysis of the broadcast problem in Chapter 3 that the adaptation of a communication pattern to the hierarchical structure improves the performance. Additionally, in the last chapter we showed that a non-hierarchical analysis ignores the possibilities of exploiting the shared-memory between processors maximally. Looking at an all-to-all pattern used in an SMP cluster, it is obvious that there are many point-to-point communications within the all-to-all pattern that are very cheap because both processors reside in the same node. Hence, the all-to-all operation should be not as expensive as expected. Motivated by this observation, we want to research if it is further possible to adapt communication patterns even more exactly to the underlying architecture in order to erase network communication completely. In the following, we will apply this technique to the problem of parallel matrix transposition that is an important operation in dense linear algebra.

In dense linear algebra, operations can be divided into three levels, see e.g. [28]. In the following, uppercase letter stand for matrices, lowercase letters stand for vectors and Greek lowercase letters are scalars. Level 1 consists of vector-vector operations, such as update ($y = y + \beta x$) or inner product ($d = y^T x$). Level 2 are matrix-vector operations, such as the matrix-vector product ($Ax = b$). Finally, level 3 accounts for matrix-matrix operations, like the matrix-matrix product ($\mathcal{C} = \mathcal{A}\mathcal{B}$). Operations of level 3 are very often expressed generally as $\mathcal{C} = \beta \operatorname{op}(\mathcal{A}) \operatorname{op}(\mathcal{B}) + \gamma\mathcal{C}$ where $\operatorname{op}(X) = X$ or X^T. An important one of these cases $\mathcal{C} = \beta\mathcal{B}^T\mathcal{A}^T + \gamma\mathcal{C}$ can be

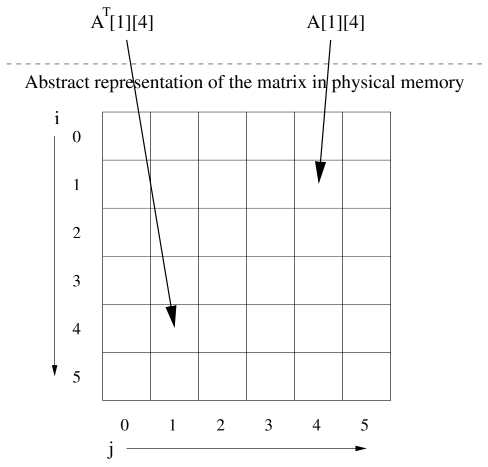


Figure 6.1: The figure depicts different access possibilities for the same physical memory location of a matrix. It is possible to interpret the matrix as transposed or not, without changing the physical representation.

implemented in 2 steps: (1) $\mathcal{T} = \beta\mathcal{A}\mathcal{B}$, (2) $\mathcal{C} = \mathcal{T}^{\mathsf{T}} + \gamma\mathcal{C}$, like for example in the PUMMA package [20].

Hence, the transposition of a matrix is a fundamental operation in dense linear algebra and is used in many scientific and engineering applications. In sequential and shared-memory programming the transposition of the matrix need not to be done in physical memory. Basically, it is only necessary to exchange the indices for the columns and the rows of the matrix. Fig. 6.1 shows the access to an element of the original matrix $\mathcal{A}$ and the transposed matrix $\mathcal{A}^{\mathcal{T}}$. We can see that it is not necessary to change the physical memory layout for these different access styles. However, depending on the number of further accesses and the physical layout of the matrix in memory, it may be better to restore the matrix in the transposed form, because a better cache utilization is possible. An overview on cache optimization techniques for numerical algorithms is given in [51].

In general, sequential and parallel shared-memory block-partitioned algorithms are used to maximize the local processor performance. Working on blocks of the matrix increases the overall performance because the memory hierarchy is used more efficiently.

This technique can also be used in a distributed-memory environment, because local computation in distributed systems does profit by that, too. However, in a distributed environment it is not possible to transpose the matrix just by exchanging the global column or row indices. Instead, communications are necessary, where processors send their blocks of the matrix to each other.

The way data is distributed among the processors in a distributed system is of fundamental importance to load balancing and communication costs. Choi and Dongarra [21] assume as target machine a non-hierarchic distributed-memory machine, where each processor has its own local memory. They use a data-distribution called *block cyclic data distribution* [52], which leads to a communication step for the transposition operation. In particular, the two-dimensional block-cyclic distribution has been suggested as a possible general-purpose basic decomposition [13]. An example for this distribution is depicted in Fig 6.2.

$$
\begin{array}{c|ccc|ccc|}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 1 & 2 & 0 & 1 & 2 \\
1 & 3 & 4 & 5 & 3 & 4 & 5 \\
\hline
2 & 0 & 1 & 2 & 0 & 1 & 2 \\
3 & 3 & 4 & 5 & 3 & 4 & 5 \\
\hline
4 & 0 & 1 & 2 & 0 & 1 & 2 \\
5 & 3 & 4 & 5 & 3 & 4 & 5 \\
\hline
\end{array}
$$

Figure 6.2: An Example for the block-cyclic distribution of the matrix: The figure illustrates the distribution from the matrix point of view, i.e. the numbers within the blocks correspond to the processors that store the corresponding block of the matrix. In this example the matrix transpose has to be done by an all-to-all communication operation.

In the example of Fig. 6.2, the matrix is divided into 6 blocks, whereby each block consists of 2×3 smaller blocks. Within each block, the smaller blocks are assigned cyclically to the 6 processors. Before and after a transposition, each processor has to store the values for the same indices of the matrix, therefore, communication is necessary. Due to the block-cyclic distribution, first, each processor has to exchange one block with each of the other processors (all-to-all communication). Second, all blocks have to be transposed locally.

A less algorithmic work was done by Haan [39]. He surveyed the problem of distributed matrix transpose technically on IBM SP supercomputers that also have the SMP cluster structure. He compared running times for the hybrid-programming model and for pure message-passing. The conclusion is that the hybrid program using MPI and OpenMP performs better than the pure MPI program.

In contrast to this technical observation, the main contribution of this chapter is to show how the communication costs of this operation can be minimized by adapting the all-to-all communication pattern to the structure of hierarchical SMP clusters. This can be achieved by defining a special data-distribution. The resulting algorithm can perform the transposition as elegant as in the sequential case.

6.1 Problem Definition

As explained in the beginning, we consider the following computation $\mathcal{C} = \beta\mathcal{A}^T\mathcal{B}^T + \gamma\mathcal{C}$. For simplicity, we assume that $\beta = 1$ and $\gamma = 0$. Hence, the considered problem is defined as follows:

- Let $\mathcal{A}$ and $\mathcal{B}$ be matrices of size $n \times m$ and $m \times n$.
- The problem is to multiply $\mathcal{A}$ with $\mathcal{B}$ and to transpose the resulting $n \times n$-matrix $\mathcal{T}$: $\mathcal{C} := (\mathcal{A} \times \mathcal{B})^{\mathcal{T}}$.
- This computation can be implemented in 2 steps. First, $\mathcal{T} := \mathcal{A} \times \mathcal{B}$, and second, $\mathcal{C} := \mathcal{T}^T$.

6.2 On-the-Fly Algorithm

Before we explain in detail how we can optimize the computation, we sketch a generic algorithm for the given problem, see *Algorithm 2*.

Algorithm 2: Generic algorithm for computing $\mathcal{C} := (\mathcal{A} \times \mathcal{B})^{\mathcal{T}}$

1. *Parallel matrix-multiplication*. The result of the parallel matrix-multiplication is the intermediate $n \times n$ matrix $\mathcal{T}$. Depending on the data distribution, each node of the SMP cluster computes one or more blocks of $\mathcal{T}$. Basically, a block of matrix $\mathcal{T}$ can be computed by a node, if it has access to the corresponding horizontal stripe of matrix $\mathcal{A}$ and the vertical stripe of matrix $\mathcal{B}$, see Fig. 6.3. As we showed in Chapter 5, where we analyzed the matrix-vector multiplication, which is a special case of matrix-multiplication, multiplying the two matrices in parallel this way does not need any communication and is therefore appropriate for distributed parallel architectures.
2. *Distribution of the blocks of $\mathcal{T}$*. After the multiplication, $\mathcal{T}$ is already distributed evenly among the nodes of the system. The distributed matrix can be transposed by communication between the nodes. Depending on the distribution of the blocks of $\mathcal{T}$, this might be even an all-to-all operation.
3. *Local transposition*. After the communication step, each processor transposes its blocks of the matrices locally.

Basically, the strategy of the algorithm should be that all pairs of blocks of $\mathcal{T}$, which have to be exchanged in order to transpose $\mathcal{T}$, were computed

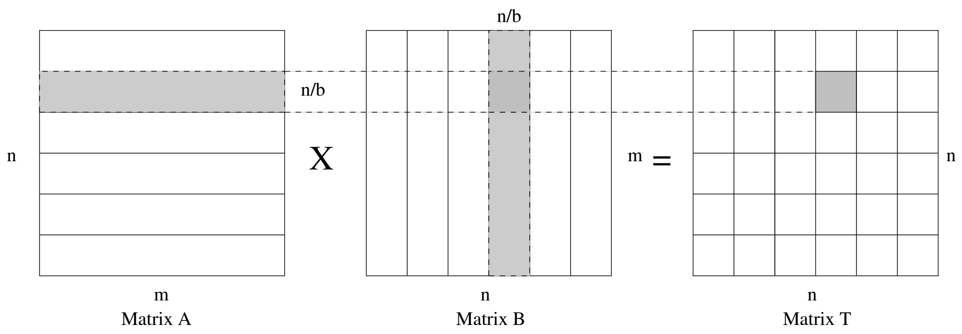


Figure 6.3: Illustration of the initial memory space for the computation of a block of $\frac{n}{b} \times \frac{n}{b}$.

by the same node and therefore, are located in its own memory. Hence, the transposition of $\mathcal{T}$ can be done *on the fly* without inter-node communication. Only local computation is necessary.

In the following, we describe the corresponding data distribution scheme called *mirror scheme*, which is a first approach to a symmetric block-distribution. In order to stay within the notion of the κNUMA model, we will denote the number of nodes in the SMP cluster by N and the number of processors per node by $\alpha(0)$.

6.2.1 Mirror Scheme

In order to realize the *on-the-fly* algorithm, we have to define which blocks of $\mathcal{T}$ have to be calculated by which node. The intermediate matrix $\mathcal{T}$ is partitioned into b^2 square blocks of equal size. We assume that b^2 is a multiple of N. Hence, each node computes $\frac{b^2}{N}$ blocks. Additionally, each node must have at least two blocks in its memory, otherwise an inner-node transpose is not possible. Thus, we assume $b^2 \geq 2N$.

- $\mathcal{T}$ is divided into b^2 blocks of size $\frac{n}{b} \times \frac{n}{b}$.
- The blocks are denoted by B_{ij}, where $0 \leq i, j < b$.
- The *block main diagonal* (*BMD*) is defined as $\{B_{ii} | 0 \leq i < b\}$.
- The *lower triangular block matrix* (*LTB*) is defined as $\{B_{ij} | 0 \leq i, j < b; i > j\}$, the *upper triangular block matrix* (*UTB*) is defined as $\{B_{ij} | 0 \leq i, j < b; j > i\}$.

The aim is that a node computes a number of pairs (B_{ij}, B_{ji}) with $0 \leq i, j < b$ and $i \neq j$ or a number of blocks on the *BMD*. If we can guarantee such a distribution for each node, then the transposition step can be done locally in each node, which is the precondition for the *on-the-fly* algorithm.

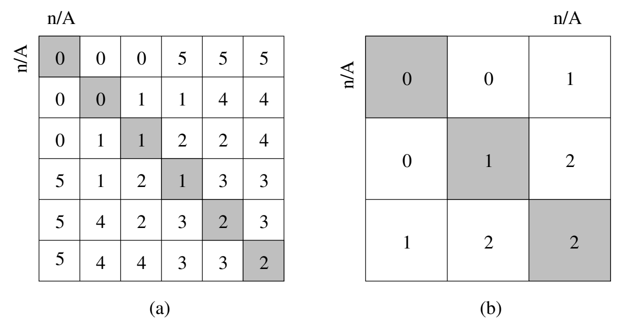


Figure 6.4: Examples for the Mirror Scheme: The numbers correspond to the nodes beginning with 0. (a) Even case: $N = 6$, b is set to 6, because $b^2 = 36$ is the value that comes closest to $2N = 12$ and fulfills the conditions $b^2 - 2N = 24 > 0$ and $b^2/N = 6$ is an integer. (b) Odd case: $N = b = 3$. The shaded regions represent the main block diagonal.

Depending on the number of SMP nodes N, there are two cases where we have to define which blocks can be computed by which SMP node.

1. If N is odd (odd case), then we set $b = N$. Each node calculates exactly one block of the *BMD*. Additionally, each node computes $(\frac{b^2}{N} - 1)/2$ pairs (B_{ij}, B_{ji}), whereby $0 \leq i, j < b$ and $i \neq j$. The idea behind this definition is that in fact, each node computes an odd number of blocks, but because one block is always on the *BMD* the remaining blocks can be pairs of blocks that have to be exchanged for a transposition. Hence, this distribution satisfies the conditions above.

2. If N is even (even case), then b is set in the way that $(b^2 - 2N)$ gets minimal, but as well greater or equal to 0. Further, b^2 has to be a multiple of N because only in this case it is possible to assign the computations of all blocks evenly among the processors. If b is determined in this way, then $\frac{b}{2}$ nodes compute two blocks of the *BMD*. Additionally, these nodes compute $(\frac{b^2}{N} - 2)/2$ pairs (B_{ij}, B_{ji}), the other nodes compute $\frac{b^2}{2N}$ pairs (B_{ij}, B_{ji}), whereby $0 \leq i, j < b$ and $i \neq j$. Again, this distribution satisfies the conditions above. The idea behind this distribution is the following. If N is even and each node must compute the same amount of blocks, then the total amount of blocks each node computes is even, too. Hence, it is necessary that the nodes compute either an even number of blocks of the *BMD* or no block of the *BMD* at all, because otherwise the remaining number of blocks for each node is odd and thus cannot be assigned to pairs of associated blocks of the UTB and LTB. On the other hand, we want to make the blocks as large as possible, because many small blocks per node may lead to an overhead in memory management. We regard the case

$b^2 = 2N$ as optimal, but this case is not possible for all values of N, because the number of blocks b has to be an integer. Therefore, we take the value for b where b^2 comes closest to $2N$.

In both cases, each node computes $\frac{b^2}{N}$ blocks of the intermediate matrix $\mathcal{T}$. An example for both cases is illustrated in Fig. 6.4.

6.2.2 Algorithm and Analysis

The *on-the-fly*-algorithm and its analysis are presented in *Algorithm 3* on page 101. An overview of the results of the analysis is depicted in Table 6.1. As presented in the analysis, the algorithm has optimal asymptotic computation and communication costs. It is possible to switch very fast from matrix $\mathcal{C}^T$ to $\mathcal{C}$, and vice versa without communication. This is an advantage for calculations, which decide at runtime, if the result of the multiplication has to be transposed or not.

Algorithm 3: On-the-fly algorithm and analysis for the computation of $\mathcal{C} := (\mathcal{A} \times \mathcal{B})^{\mathcal{T}}$

1. **Parallel Matrix-Multiplication.** Each node computes $\frac{b^2}{N}$ blocks of size $\frac{n}{b} \times \frac{n}{b}$ with $\alpha(0)$ processors. This can be done in

$$O\left(\frac{\frac{b^2}{N}(\frac{n}{b}m\frac{n}{b})}{\alpha(0)}\right) = O\left(\frac{n^2 m}{p}\right) \text{ steps}$$

2. **No Communication.** According to the mirror scheme, no communication is necessary, since B_{ij} and B_{ji} are located in the same node.

3. **Parallel Transpose.** In each node $\frac{b^2}{N}$ blocks of size $\frac{n}{b} \times \frac{n}{b}$ have to be accessed by $\alpha(0)$ processors. This can be done in

$$O\left(\frac{b^2}{N}\frac{n^2}{b^2}\frac{1}{\alpha(0)}\right) = O\left(\frac{n^2}{N\alpha(0)}\right) = O\left(\frac{n^2}{p}\right) \text{ steps}$$

The only sub-optimal feature is the initial memory space. The *initial memory space* is the number of elements of the matrices $\mathcal{A}$ and $\mathcal{B}$, which have to be stored in the local memory of the nodes in order to perform the multiplication. It would be optimal to store $2\frac{nm}{N}$ elements per node or $2\frac{nm}{p}$ elements per processor. However, the algorithm for parallel matrix-multiplication needs more elements per processor, depending on the distribution of the computation of the blocks.

On-the-Fly Algorithm	Complexity
Matrix-Multiplication	$O\left(\frac{n^2 m}{p}\right)$
Matrix-Transpose	$O\left(\frac{n^2}{p}\right)$
Communication	0
Initial Memory Space per processor	$\leq b2\frac{nm}{p}$

Table 6.1: Analysis of the On-the-Fly algorithm.

Until now, the *mirror scheme* does not define exactly, which blocks are calculated by which nodes. We only know the number of blocks each node has to compute and the sizes of the stripes of the initial matrices that are necessary for each block separately. Hence, we can only give an upper bound. For each block, at most $2\frac{nm}{b}$ elements of the matrices $\mathcal{A}$ and $\mathcal{B}$ are necessary (see Fig. 6.3). Therefore, each node has to store $\frac{b2nm}{N}$ elements, which means, that it depends on the number of blocks, if this value comes close to the optimum or not. Each processor stores $\leq \frac{b2nm}{p}$ elements. Further, it is obvious, that $N \geq b$, because no node has to store more than both matrices ($2mn$). Hence, obviously $2mn$ (both matrices) is the upper bound.

Fig. 6.5 is an example that shows that the size of the initial memory space depends on the distribution of the blocks to the nodes. In the example, the same number of blocks is computed by each node. While in the left case only a part of the initial matrices have to be stored, in the right case the node has to store both matrices completely.

In Section 6.3, we will improve the lower bound of $2\frac{mn}{N}$ for the initial memory space of a node. Further, a general exact mapping of blocks, called *snake-like scheme*, is presented. The initial memory space of this scheme is much lower than the upper bound and comes close to the lower bound.

6.3 Reducing the Initial Memory Space

The initial memory space is important, because it is an indicator for work that has to be done before the needed data can be used by the nodes. The data does not get there for free, either it gets there by communication or by reading data from a storage medium. In both cases, the more data is necessary, the more time is wasted. Further arguments are that in general programs are faster if they use a smaller amount of memory, and in large-scale applications, memory is also a critical resource, which has to be used carefully.

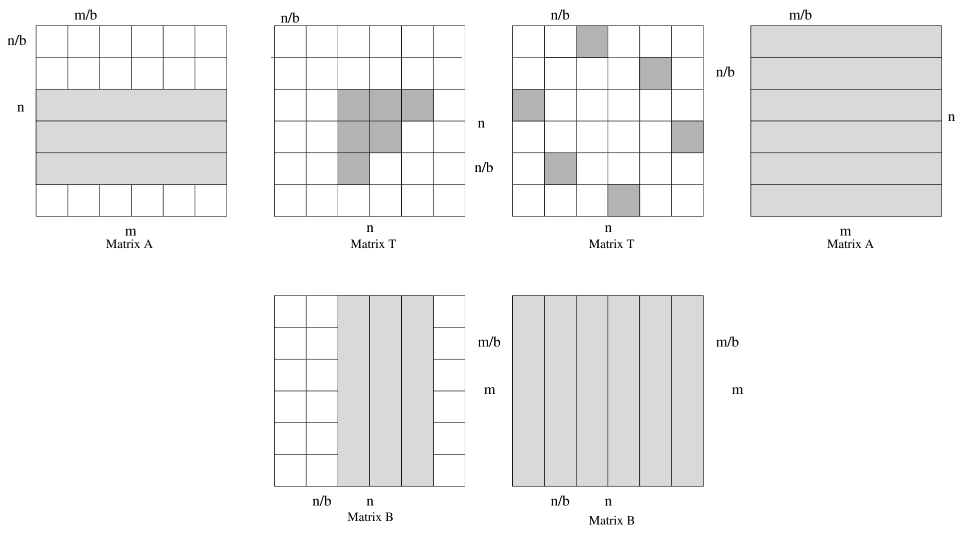


Figure 6.5: An example for different initial memory space, depending on which block is calculated by which node. In the left case, a node needs only $6\frac{nm}{b}$, while in the right case it needs $12\frac{nm}{b}$. In this example $N = b = 6$, this means that one node has to store both matrices, while the other has to store only the amount of one matrix.

6.3.1 Lower Bound

As mentioned above, a distribution of the matrices $\mathcal{A}$ and $\mathcal{B}$ before the multiplication over the N nodes that leads to store only $\frac{2nm}{N}$ elements in each node can be regarded as optimal concerning the initial memory space. The *on-the-fly* algorithm avoids communication operations for the parallel matrix-multiplication and hence, it is necessary to store more elements of the initial matrices per node. In order to compute a block of $\mathcal{T}$, the node needs a horizontal stripe of $\mathcal{A}$ and a vertical stripe of $\mathcal{B}$ with the appropriate sizes, see Fig. 6.3. If a node has to compute more than one block, it is possible for the node to reuse stripes from $\mathcal{A}$ or $\mathcal{B}$, if another block of the same block-column, or block-row, is computed by the node, too. The best reuse is achieved, if all blocks, computed by one node, build a bigger square block around the *BMD*. Basically, it is not possible to define a distribution, which satisfies this assumption for all nodes ($N > 1$). Therefore, the initial memory space for this situation is a lower bound for all data-distributions. If the $\frac{b^2}{N}$ blocks of size $\frac{n}{b} \times \frac{n}{b}$ are arranged quadratically, then the initial memory space for one node in this case is the best lower bound and can be charged as

$$2\left(\sqrt{\frac{b^2}{N}}\frac{nm}{b}\right) = 2\left(\frac{b}{\sqrt{N}}\frac{nm}{b}\right) = 2\frac{nm}{\sqrt{N}}$$

6.3.2 Snake-like Scheme

The *snake-like scheme* extends the *mirror scheme* by a method how the blocks of the *LTB* and the *UTB* are distributed among the nodes. The running time of the algorithm stays the same, because each node computes the same number of blocks as in the mirror scheme. Basically, the snake-like scheme defines exactly which blocks have to be computed by which node in order to reduce the initial memory space per node.

The blocks of the *BMD* are distributed as described in the *mirror scheme*. In the following, we will only describe how to map the blocks of the *LTB*. The mapping for the *UTB* can be received by exchanging the block indices. The scheme is defined by a ranking of blocks of the *LTB* and by a ranking of nodes. Each node has a unique number from 0 to $N-1$. Each block in the *LTB* has a number from 0 to $\frac{b^2-b}{2}-1$, and the blocks of the *BMD* have the numbers 0 to $b-1$. There are two steps, first, the blocks of the *BMD* and second, the blocks of the *LTB* are assigned to the nodes. The mapping starts by assigning blocks to nodes with increasing node numbers and increasing block numbers.

1. The blocks of the *BMD* are assigned to the nodes as follows. If N is even, then the first $b/2$ nodes get two blocks of the *BMD* with increasing block numbers. If N is odd, then each node gets the block of the *BMD*, where the block number is equal to the node number.

2. The blocks of the *LTB* are assigned to the nodes as follows. We start again with node 0 and continue with increasing node numbers. Each node gets its remaining number of blocks according to the numbering of the blocks in the *LTB*. The number or rank of a block in the *LTB* is defined next.

Let $R(i,j)$ be the rank of the block B_{ij} in the *LTB*. $R(i,j)$ is defined by

$$R(i,j) := \sum_{k=0}^{x}(2b-3-4k) + (1-y)(z-2(j+1)) + y(z+2(i-b))$$

, with
$$x = ((i-j) \text{ div } 2 + (i-j) \bmod 2) - 1$$
$$y = ((i-j) \text{ div } 2 + (i-j) \bmod 2) \bmod 2$$
$$z = (i-j) \bmod 2$$

The resulting ranking for the blocks in an *LTB* is presented in Fig. 6.6 for $b = 6$. The snake-like scheme is shown in Fig. 6.7. In the following, we adhere the properties of the snake-like scheme:

- First, the *snake-like scheme* guarantees a symmetric and even distribution of the blocks among the processors.

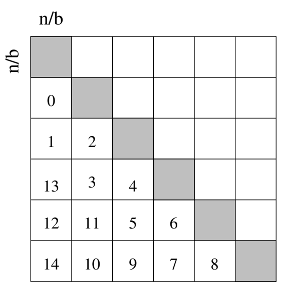


Figure 6.6: Snake-like scheme. Ranking of the blocks of the *LTB*. The numbers represent the block numbers.

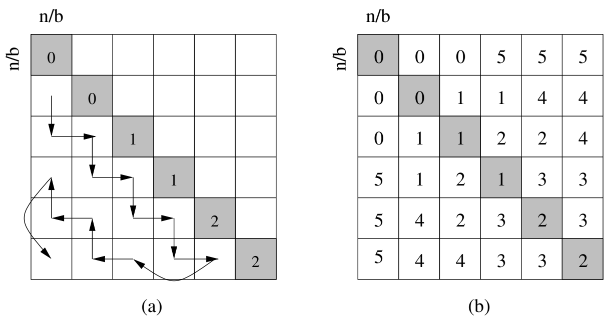


Figure 6.7: Snake-like scheme. The numbers represent the node numbers. (a) The mapping of the *BMD*-blocks was already done, and the arrows show the order in which the blocks will be mapped to the nodes. (b) The resulting snake-like scheme for $N = b = 6$. In this case, the numbers denote which node will computed the respective block.

- Each block that resides in the *LTB* (or *UTB*) and was computed by the same node has one common stripe of the initial matrices with at least one other block computed by the same node.
- By starting the distribution of the blocks of the *LTB* (*UTB*) with the nodes that have to calculate blocks of the *BMD*, the scheme automatically creates a good locality of all the blocks of these nodes. In some cases, the blocks of the *BMD* of a node share at least one stripe with the blocks of the node, which are not on the *BMD*.

In the following, the blocks that are computed by the same node in the *LTB* are denoted by *lower group*, and the blocks in the *UTB* are denoted by *upper group*. Further, we denote a group of blocks or a block on the *BMD* as *isolated*, if it shares no stripe (horizontal or vertical) of the initial matrices with another group or block within the node. Basically, due to the snake-like scheme the lower and the upper group of each node are always isolated.

Maximal Memory Usage in Best- and Worst-Case Scenarios

Now we formulate best and worst-case behaviors. More exactly, we are interested in the maximal amount of initial memory space that nodes have to work on in a best- and a worst-case scenario. This analysis gives the information about the minimal and the maximal memory requirements of the nodes. In the following, we have to distinguish again the odd and the even case.

1. If N is even, the best-case scenario for the nodes, which compute two blocks on the *BMD*, is that each block shares its vertical stripe with the upper group and the horizontal stripe with the lower group. The lower and upper group are always isolated from each other. Hence, the nodes which do not compute blocks on the *BMD*, are the bottleneck, because their lower and upper groups consist of one more block. Hence, these nodes will have the highest memory usage in a best-case scenario. Under the assumption that each block in the lower group shares one stripe with at least one other block of the group, only 2 stripes of the initial matrices are necessary for the first block. For all other blocks, only one more stripe is necessary that is not yet known. Because each node that does not compute blocks on the *BMD* has $b^2/2N$ in the lower as well as in the upper group, the initial memory space in this case can be charged as

   $$2\left(\frac{2mn}{b} + \left(\frac{b^2}{2N} - 1\right)\frac{mn}{b}\right) = \frac{2mn}{b} + b\frac{mn}{N}$$

   In the worst-case, if N is even, the lower and upper group are isolated from each other and from the two blocks on the *BMD*. In this case, we can calculate the initial memory usage in the following way. Four stripes are necessary for the two blocks on the *BMD*, for the remaining blocks the number of stripes can be calculated similar to the best case.

   $$\frac{4mn}{b} + 2\left(\frac{2mn}{b} + \left(\frac{\frac{b^2}{N} - 2}{2} - 1\right)\frac{mn}{b}\right) = \frac{4mn}{b} + b\frac{mn}{N}$$

2. If N is odd, then all nodes are in the same situation. They compute one block on the *BMD*, and the remaining blocks are part of the lower and upper group. Therefore, in the best case the block on the *BMD* is not isolated from the lower and upper group.

   $$2\left(\frac{2mn}{b} + \left(\frac{\frac{b^2}{N} - 1}{2} - 1\right)\frac{mn}{b}\right) = \frac{mn}{b} + b\frac{mn}{N}$$

As mentioned above, in this case $b = N$ and therefore the initial memory space is:

$$\frac{mn}{N} + mn$$

In the worst case, where the block on the *BMD* is isolated, we get an initial memory space of

$$3\frac{mn}{N} + mn$$

In Table 6.2 the bounds are summarized.

	Best Case	Worst Case
even	$\frac{2mn}{b} + b\frac{mn}{N}$	$\frac{4mn}{b} + b\frac{mn}{N}$
odd	$\frac{mn}{N} + mn$	$\frac{3mn}{N} + mn$

Table 6.2: Analysis of the snake-like scheme.

6.3.3 Optimizing the Number of Blocks

In the odd case, the size of b is already defined by the *mirror scheme* ($b = N$). In the even case, we can minimize the functions for the initial memory space above and resolve them by b[1]. In the best case the initial memory space gets minimal, if $b = \sqrt{2N}$ is a valid solution. Since the function is monotonic increasing for $b \geq \sqrt{2N}$, b has to be chosen as small as possible. However, this is guaranteed by definition of the mirror scheme in Section 6.2.1 on page 99.

6.3.4 Comparison of the Bounds

Now, we want to compare the received bounds for the initial memory space with each other and with the lower and upper bound defined above. The odd case is depicted in Fig. 6.8 and the even case in Fig. 6.9.

In the odd case, the initial memory space per node decreases with an increasing number of nodes N. In the worst case, as well as in the best-case, the value converges towards mn, but will never reach it. Compared with the upper bound ($2mn$), the value converges towards the half of the upper bound value. The distance to the lower bound slightly increases with increasing values for N.

The values for the maximal initial memory usage in the odd case are either equal to the worst case or equal to the best-case, because b is static.

[1] The minimization of the best case expression can be done as follows: $(\frac{2mn}{b} + b\frac{mn}{N})' = -\frac{2mn}{b^2} + \frac{mn}{N} = 0 \rightarrow b = \sqrt{2N}$.

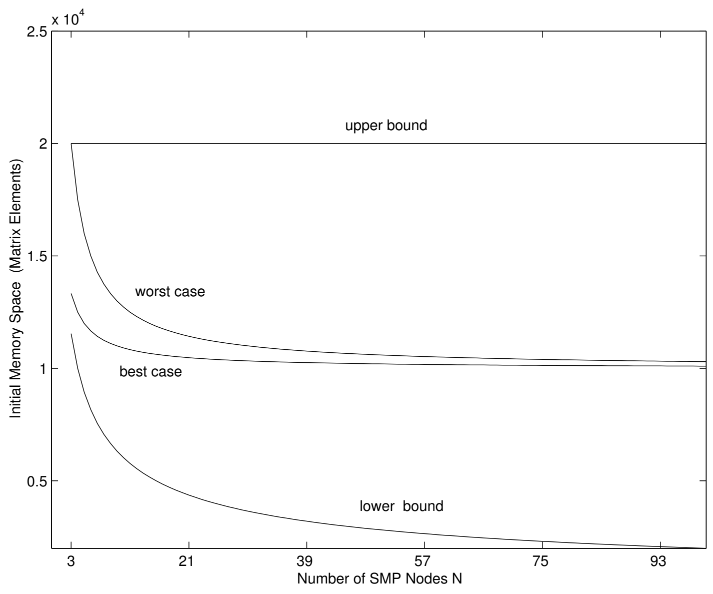


Figure 6.8: Each node computes an odd number of blocks. The initial memory space of a certain N is either on the best case line or on the worst case line. In the example a matrix consists of 10000 elements.

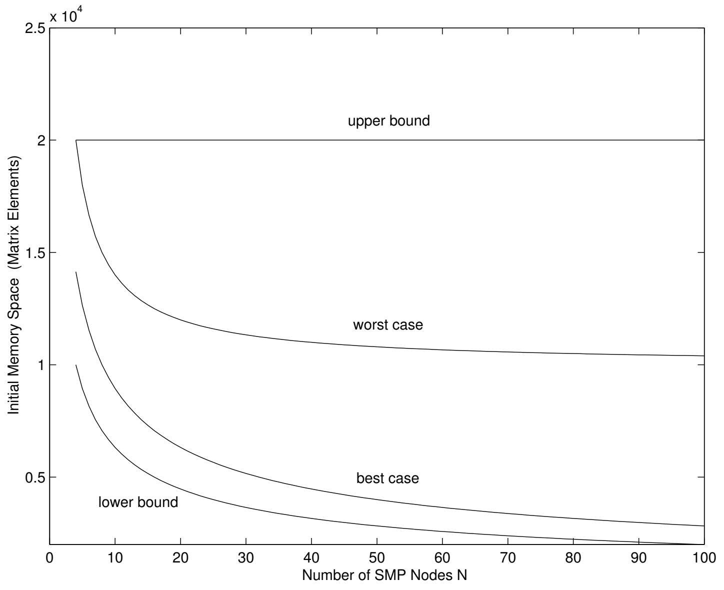


Figure 6.9: Each node computes an even number of blocks. The initial memory space of a certain N is either on the best case line or on the worst case line or in-between. In the example a matrix consists of 10000 elements.

In the even case, the spectrum of possible values is even bigger, because the number of blocks b is not fix. In the best case scenario, we get the best possible result for $b = \sqrt{2N}$. In the worst-case scenario, the worst result is achieved for the highest value of b possible. The highest value that always fulfills the requirements of the mirror scheme is $b = N$.

Hence depending on the number of nodes, the initial memory space is a value between $2\sqrt{2}\frac{mn}{\sqrt{N}}$ and $4\frac{mn}{N}+mn$. In the best-case scenario, we are very close to the lower bound. It is only a factor of $\sqrt{2}$ worse. In the worst-case, the initial memory space converges towards mn with increasing number of nodes.

6.4 Summary

The main aspect of the chapter is to show a method for algorithmic optimizations on SMP clusters based on the idea of reducing communication costs by adapting communication patterns to the hierarchical structure of SMP clusters. As an example for the application of the technique, we presented an algorithm for computing an $n \times n$-matrix $\mathcal{C}$ by , first, multiplying an $n \times m$-matrix $\mathcal{A}$ with an $m \times n$-matrix $\mathcal{B}$ and, second, transposing the resulting $n \times n$-matrix $\mathcal{T}$. The algorithm is optimal concerning asymptotic computation time. Due to special data-decomposition (*mirror scheme*), it is possible to perform the transpose without any communication operation. Therefore, the algorithm is improved concerning communication costs, because the all-to-all communication step was eliminated.

Another criterion observed by the analysis is the initial memory space needed by each processor before the computation. We presented a lower bound for the initial memory space of the *on-the-fly* algorithm, and the *snake-like scheme* that comes close to the lower bound (factor $\sqrt{2}$) in some cases. For all cases, the initial memory space converges with an increasing number of nodes towards nm, which is the size of one initial matrix. This is only half of the worst-case behavior.

The main conclusion is, that the presented method is another and better way to optimize algorithms for SMP clusters than just migrating existing message-passing algorithms directly. In the presented algorithm, the key for the optimization of the communication pattern was the data decomposition. Because of a special decomposition, the existing communication pattern became optimal. There are probably more problems in dense linear algebra or other areas, where such a strategy may be successful. As presented in the analysis, the efficiency of the decomposition depends on the structure of the underlying architecture, or instance of the architecture. Hence, parallel programs and libraries should take the parameters (like N, $\alpha(0)$) of the target system into account.

Chapter 7

Hierarchical Sensitive Design

The last method we want to present in this thesis, is called *hierarchical sensitive design* for algorithms. In the beginning of the thesis, we showed the technological trend to network and communication hierarchies. We explained why in future the number of hierarchies will probably grow and that communication over higher levels of the hierarchy is more expensive than over lower levels. Under this assumption, algorithms that try to minimize communication and try to shift the complexity to local computations will have great relevance for future architectures. Algorithms that try to decompose and distribute the data in a pre-processing step are thus very suitable. Communication only takes place in this step, afterwards only local computation is necessary to produce the results. As an example for the development of such algorithms, we review a sorting method called *Radix Sort* [68, 50]. Radix Sort is a method to sort a set of integer keys using their binary representation. We present the various algorithms from the sequential to the latest parallel algorithm and we describe the motivations for the respective improvements.

7.1 Sequential Radix Sort

The main idea of the algorithm is to sort the keys in several iterations using parts of the binary representation. In each iteration, the keys are sorted according to a certain number of bits (*radix* of size $\mathtt{r}$), starting in the first iteration with the least significant bit. Assuming the keys consist of $\mathtt{l}$ bits, the algorithm needs $\mathtt{l}/\mathtt{r}$ iterations to sort the keys. In order to introduce the algorithm, we assume that the keys are stored in an array `S` (Source). After the iteration, it will be stored in an array `D` (Destination). After each iteration, the arrays exchange their roles. Further, only one working array H (Help) of size $2^{\mathtt{r}}$ is necessary to store the histogram of keys. An example is depicted in Fig. 7.1.

Algorithm 4: Outline of the sequential straight radix sort algorithm.
In each of the `l/r` iterations, the following 3 steps have to be done

1. *Generation of key histogram.* Scan all keys in `S`. For each key, compute the value v of the actual `r` bits. Increase the value of `H[v]` by one. After scanning all keys, the array `H` contains the number of keys for each possible value of the actual `r` bits.
2. *Computation of prefix-sums.* Now, the array `H` is turned into an array containing the starting indices for each of the 2^r buckets. This can be done by creating the prefix sums on `H`.
3. *Moving keys from source to destination.* Again all keys have to be scanned. For each key `i`, we compute its new index in array `D`. This can be done by first computing the value v of the actual `r` bits. This value can be used as an index in the prefix-sum array `H`. The corresponding value at this index in the array `H` is the new index of the key in the array `D` (`D[H[v]]=S[i]`). Next, the value at the corresponding index in the array `H` is increased by 1.

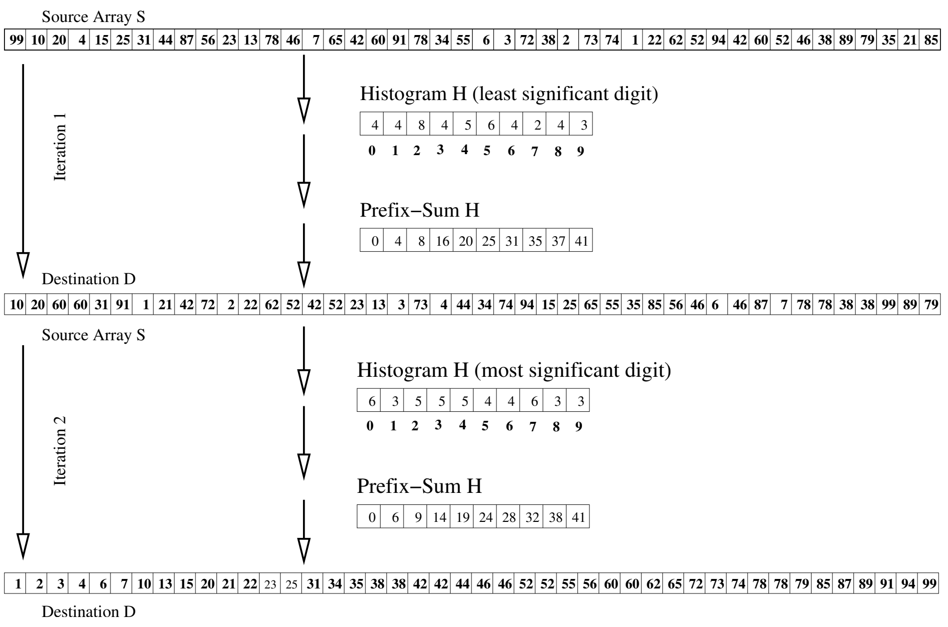


Figure 7.1: An example for the sequential straight radix sort algorithm. In each iteration, a radix of one decimal digit is used to sort keys consisting of two decimal digits. Hence, the data array can be sorted in two iterations.

It is not easy to see, why this algorithm really sorts the input values. Nevertheless, it works, because it is guaranteed that each iteration is stable. That means that two keys only change their relative position, if one radix is different. Otherwise, the order remains. Hence, it can be shown easily by induction that the algorithm is correct. Detailed analyses of this sequential radix sort algorithms can be found in standard textbooks, see e.g. [68, 50], where it is called straight or straightforward radix sort.

7.2 Parallel Radix Sort Algorithms

In the next sections, we will present several parallel algorithms in historical order for Radix Sort that are suitable for distributed-memory machines. Anyhow, each algorithms has certain drawbacks concerning either communication costs or load balancing. We will show how these drawbacks can be eliminated step-by-step.

7.2.1 Straight Parallel Radix Sort

A straightforward parallelization of radix sort is not very different to the sequential algorithm. Initially, the keys are distributed equally among the processors. The main idea of the parallelization is that the processors are viewed as buckets or groups of buckets. In each of the l/r iterations each processor creates $B = 2^r$ buckets. The buckets are assigned evenly to the processors, i.e. each processor corresponds to B/p buckets. Each iteration of the algorithms consists of three steps.

Algorithm 5: Description of the steps of each iteration of the parallel radix sort algorithm

1. *Creation of local buckets*. Each processor scans its local keys with a certain radix r and stores them in the corresponding buckets.

2. *Communication of bucket sizes*. The sizes of the locally created buckets are exchanged by the processors in order to calculate a communication pattern for the buckets. In general, this is an all-to-all communication, but each processor has to know how large the receiving buckets will be.

3. *Communication of the buckets*. The keys are exchanged according to the computed communication pattern. Each processor receives its buckets from all the other processors.

These steps are repeated until all bits of the keys were scanned, i.e. after l/r iterations. More detailed descriptions of radix sort algorithms for shared-memory and distributed-memory machines can be found in [6, 46].

Obviously, this algorithm has two main drawbacks. First, depending on the sizes of l and r there are several iterations, which all contain two all-to-all communication operations, one for the bucket sizes and one for the buckets themselves. The communication of the bucket sizes is not very expensive, because the amount of data is low. However, the communication of the keys might become very expensive if n is large enough.

Second, the assignment of buckets to processors leads to load imbalance of the processors. Usually the sizes of the buckets will vary a lot, because in general the values of the keys are not distributed ideally. Hence, with each iteration the imbalance between the processors will grow which leads to an inefficient load. In the following, we will show improvements for each of these drawbacks.

7.2.2 Load Balanced Parallel Radix Sort

One main problem with this approach is its irregularity in communication and computation, which arise because of data characteristics like e.g. data skew or duplicates. Therefore, in [70] a load balanced parallel radix sort is presented. This algorithm splits the locally generated buckets with respect to balance the number of keys that have to be sent to each processor. Therefore, the resulting communication step is a real balanced all-to-all operation between the processors.

The algorithm works in the same way as the last one, except that it does not stick exactly to the assignment of buckets to processors. After the processors exchanged the sizes of the respective buckets, each processor can compute balanced parts of the keys by splitting the buckets. Thus, it is possible that a processor sends one part of a local bucket to a processor and the rest to another one. The example illustrated in Fig. 7.2 assumes four processors, and hence, in each iteration four buckets are built locally (radix $r = 2$). Further, in the beginning, each processor stores 10 integer keys. The aim is to split the buckets in the way that each processor receives exactly 10 integer keys (including its own). The splits can be computed easily by each processor and are depicted in Fig. 7.2b.

This algorithm provides that the load is balanced, but the problem of having too much communication cost still remains.

7.2.3 Hierarchical-Sensitive Design

All presented algorithms perform the Radix Sort starting with the least significant bits. It follows directly from the approach that the order in intermediate iterations has not necessarily to do anything with the final order.

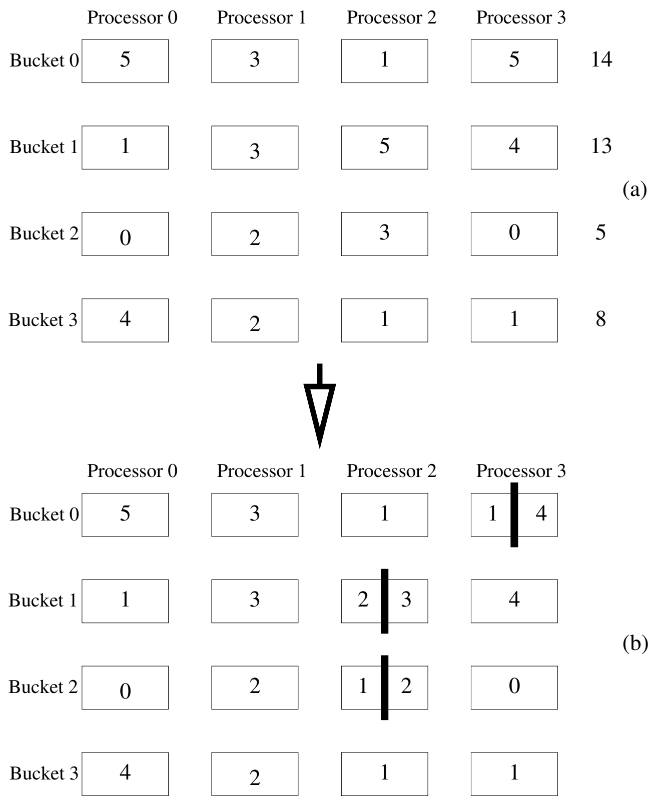


Figure 7.2: An Example for illustrating the idea of load balanced radix sort. The splitting of the buckets is shown by the vertical black bars. For example, processor 0 receives all *bucket 0* from processor 1 and 2, but only one key from the *bucket 0* of processor 3.

In each iteration, bits that are more significant are considered which may lead to the fact that lowest value gets highest one and vice-versa. Hence, in a worst case scenario, the integer keys have to be communicated over the highest level of the communication hierarchy of a κNUMA machine in each iteration. Let us consider a simple example for this situation. The target machine is a regular κNUMA machine that consists of 8 processors, with $\kappa = 2$ and $\alpha(0) = \alpha(1) = \alpha(2) = 2$, see Fig. 7.3. There are 8 keys and we assume their binary representation has length $l = 9$ and radix $r = 3$. Hence, the algorithm needs 3 iterations to sort the keys. The keys are selected according to the following rule. We need 3 binary digits to code the processor numbers. Before the first iteration, each processor stores a key where the 3 most significant bits are equal to the inverted processor number, the 3 bits in the middle are equal to the processor number and the 3 least significant bits are again equal to the inverted processor number. Thus, processor 7 ($= 111_b$) stores the integer 52=($= 000111000$) before the first iteration. Starting with the least significant bits leads to the fact that in every iteration the keys have to be sent over the highest level of the network, because the keys have to be sent to the processor with the inverted processor number and this is always a processor with maximal distance.

Hence, the idea is to sort the keys by starting at the most significant bits. The main characteristic of buckets produced by this approach is that the order among the buckets will never change again in the algorithm. Only the keys within the buckets have to be sorted further. Hence, we could try to divide continuously the input keys among the sub-machines. Further iterations will only take place within the sub-machines, and by that, the level over which communication takes place is also reduced with every iteration.

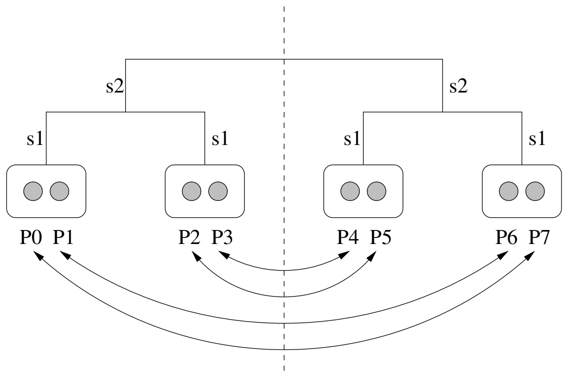


Figure 7.3: A sample architecture illustration of a non-hierarchical sensitive algorithm. The architectural parameters are set to $\kappa = 2$, $\alpha(2) = \alpha(1) = \alpha(0) = 2$. The arrows depict the communication partners.

However, with such a solution we still have to communicate the keys several times. Although the level of communication decreases in every iteration, it would be much better if it were possible to distribute them only once. This is possible, if we can find a balanced partition of the input data with respect to the processors. Due to such a partition, it is possible to send the keys only once and to sort them in a second step locally. Concerning our example, the partitioning is received by sorting the keys according to the 3 most significant bits. The local sorting step is not necessary in this case, because there is only one key per processor. In the following section, we will present algorithms that work according to that principle.

7.2.4 Communication Sensitive Parallel Radix Sort

Until now, all mentioned algorithms need l/r key- and counter-communication steps. Especially for distributed-memory machines, where the communication is done over an interconnection network, this might be very time consuming. The communication and cache conscious radix sort algorithm (C3-Radix) [45] tries to improve this situation by starting the radix sort at the most significant bit. The intention of C3-Radix is to partition the data into several buckets which can be distributed equally among the processors. Therefore, the choice of the radix is very important. If the distribution of the buckets is not possible in a balanced way, the radix is enlarged and more fine-grained buckets are created. This is repeated until a good load-balance is achieved. Then the keys are sent with an all-to-all operation among the processors. The remaining task is to sort the buckets locally. For this step, the authors use a very efficient cache conscious radix sort algorithm, which uses the fact that the data is already sorted for a certain number of bits. The algorithm has to communicate the keys only once, but because of the characteristics of the data, there might be several iterations where the bucket counters have to be exchanged. Depending on the number of iterations and the size of the radix, this can increase the running time of the algorithm extremely.

The algorithm sorts n keys and each key consists of l bits. The used radix has length r. Initially, each processor stores n/p keys. The keys are globally sorted if the keys within each processor are sorted, and there is a known order of the processors for which all keys of one processor are greater or equal to all keys in the preceding processor. Each processor builds buckets of keys by starting to observe the first r bits of each key. The initial length of the radix should be chosen in the way that $2^r > p$. Keys with the same radix belong to the same bucket. All steps of the algorithm are explained in *Algorithm 6* on page 118.

7.2.5 An Alternative Approach: Sample Sort

Another approach is based on *Sample Sort*, which is often used in parallel sorting; see e.g. [41, 33, 14, 69]. Each processor samples q keys from its n/p keys and exchanges them with the other processors. Sorting the set of samples in each processor makes it possible to create a set of $s - 1 < q$ equidistant keys called splitters. These splitters can be used to create s buckets of approximately size n/s. Load balance can be achieved, if s is a multiple of p. The parallel counting split radix sort algorithm (PCS-Radix) [47] uses Sample Sort for the partitioning of the data instead of using the radix of the keys directly. Radix sort is only used to sort the buckets locally. The prize for the independence of the data characteristics is the detection of global samples in the distributed system, but in a badly distributed environment, this investment is worth doing. Despite of that, in environments where the data is mostly well distributed, the C3-Radix algorithm should be the algorithm of choice, because each processor is able to start the local creation of the buckets directly, and in general, no further iterations are necessary to build the local buckets. On the other hand, also in a well-distributed

Algorithm 6: Outline of the C3-Radix algorithm

1. *Reverse Sorting*. Each processor scans its n/p integer keys using the first r bits beginning with the most significant bit, and building the corresponding 2^r buckets. During the creation of the buckets, a counter array is built, too. Each entry in the counter array contains the number of keys in the corresponding bucket.

2. *Communication of Counters*. The local 2^r counters are exchanged between the processors. After this step, each processor knows the total amount of elements per bucket.

3. *Computation of bucket distribution* . Each processor computes locally a distribution of the buckets to the processors. If it is not possible to achieve a good load balance, then each processor starts again with step 1 and sets the new radix to ir, where i is the number of iterations. By extending the radix, the algorithm tries to produce more and smaller buckets that may lead to a better load balance. Otherwise, the algorithm continues with step 4.

4. *All-to-All key communication*. The buckets are sent in an all-to-all fashion. After this step, no more communication is necessary.

5. *Local sorting*.

environment, sometimes there may arise badly distributed data sets. The C3-Radix algorithm does not have the capability of being efficient in these cases, but it should be guaranteed that it is not far away from being efficient.

Hence, in the following we survey the possibilities for C3-Radix to be more stable and more predictable working on unbalanced data distributions. In Section 7.3, we will explain the problems of C3-Radix in such cases more deeply, and we will give an example. In Section 7.4 we will give and interpret results of experimental test, and in Section 7.5 we conclude.

7.3 Further Improvements of the Communication Sensitive Parallel Radix Sort

In the last sections, we described that the parallel versions of Radix Sort suffer either from too much communication costs or from unbalanced data that leads to a bad load balance among the processors. With the following improvements for the C3-Radix algorithm,, we want to eliminate both drawbacks at once. As we can see in Section 7.2.4, the main problem with the C3-Radix algorithm is that the first 3 steps may have to be repeated several times. The number of iterations depends on the data distribution, the initial radix chosen and the size of the integer keys. The more necessary iterations, the larger the radix. Since the number of buckets as well as the number of counters is 2^r, the allocated memory and the amount of data that has to be communicated may increase the running time of the algorithm tremendously. What we want to improve is the way the algorithm tries to distribute the data equally among the processors, if more than 1 iteration is necessary. All other optimizations of the C3-Radix algorithms will not be changed.

In order to explain the problem more detailed, we give an example. We want to calculate the total number of counters that have to be communicated during the execution of the C3 algorithm. Let i be the number of iterations. In iteration 0, the first three steps of the algorithm are executed for the first time. Hence, the accumulated number of counters over all iterations for each processor can be charged as

$$\sum_{j=0}^{i} 2^{(j+1)r}$$

In the experiments of [45] the radix is set to 5. If we assume there is a data distribution that leads to 6 iterations ($0 \leq j \leq 5$), where in the last iteration 30 bits have to be compared, then the total number of counters broadcasted over the network is $1,108,378,656$. If we further assume that a counter is a 32 bit integer (which is necessary for a large n), then we see that each processor has to communicate about 4.129 GB of data. Each processor sends its counter array to all the other $p-1$ processors. Assuming 16 processors, the total data transferred by the network is 990.96 GB.

Besides this communication problem, there might arise memory problems. At least each processor has to store the counter array and the buckets constructed locally. Although there are more data structures like for example the initial data (n/p elements) or different counter arrays for the communication operation, the memory demand is dominated by the counter array. Therefore, we take its estimated memory demand as a key figure. Since the counter array has size 2^{30} in the sixth iteration and each entry stores one 32 bit integer, each processor needs at least 4 GB of main memory. Depending on the implementation of the counter communication (allreduce, broadcast, ..) the node has to buffer up to p counter arrays which leads to 64 GB in our example. For a huge part of supercomputers, namely the PC-based SMP- or workstation-clusters, this size is not manageable. The program will abort due to out of memory errors. In order to avoid these large data arrays, our idea is not to rebuild all the buckets and counters with a larger radix within each iteration, but only to rebuild those that are necessary. The algorithm of choosing the buckets that have to be rebuilt needs the 3 steps explained in Algorithm 7 (page 120).

Algorithm 7: Algorithm for detecting buckets to rebuild in further iterations

1. Build a prefix array `count` using the actual global counter array, which is known after the broadcast of the local counters. The first entry of the prefix array is the overall number of elements in the first bucket. The second entry contains the sum of the first entry and the number of elements in the second bucket. And so forth.
2. The optimal number of integer keys each processor should have before the local sorting step is n/p. We build a prefix array `proc` for the optimal number of keys per processor. The first entry contains n/p, the second entry contains $2n/p$, and so forth. The whole array has p entries.
3. For all entries in `proc`, we search the index in `count` where the value of `proc` is lower than the value of `count`. All these indices are collected and only the buckets and counters with these numbers are rebuild.

In the first iteration, the algorithm works as described in Section 7.2.4. The only difference is that after checking if another iteration is necessary, the buckets, which have to be rebuilt are detected with the method above. For further iterations the first 3 steps are replaced by the 3 steps in Algorithm 8 on page 120.

Algorithm 8: Changed steps for the communication conscious algorithm

1. *Reverse Sorting of special buckets.* Rebuild the buckets and counters detected by the above method with a larger radix.
2. *Communication of Counters.* All-to-all communication of the new counter array.
3. *Computation of bucket distribution.* Check, if another iteration is necessary. If it is necessary, detect the buckets that have to be rebuilt with a higher radix. Otherwise, proceed to step 4 of the algorithm described in Section 7.2.4.

Now, we want to analyze the behavior of this new algorithm in order to judge the received improvement. Hence, we calculate the same numbers as for the original C3 algorithm. In each iteration, a certain number of buckets have to be rebuilt. We denote this number with a_j, where $0 \leq a_j < p$ and $j > 0$ is the number of the iteration ($a_0 = 0$). Hence, in iteration i, the size of the counter array is

$$2^r + \sum_{j=0}^{i} (a_j 2^r - a_j)$$

The accumulated amount of counters for each round and for each processor assuming that i iterations are performed can be charged as

$$\sum_{j=0}^{i} (2^r + \sum_{k=0}^{j} (a_k 2^r - a_k)) = i2^r + (2^r - 1) \sum_{j=0}^{i} \sum_{k=0}^{j} a_k$$

Looking at the scenario above ($r = 5, i = 5, p = 16$) and assuming the worst-case, where $a_j = p - 1$ for $j > 1$ the over-all number of counters broadcasted over the network by one processor is 7167. Again, we assume that a counter is a 32 bit integer, and then we see that each processor has to communicate about 27.996 KB. In each iteration, each processor sends its counter array to the other $p - 1$ processors. Hence, the overall amount of counters transferred by the network is $6,719.063$ KB. Compared to the 990.96 GB of the not improved version, this is negligibly small.

The situation is also much better for the memory requirements. Storing the counter array of size 2357 in iteration 5 requires 9.21 KB per processor, which normally is negligible compared to custom computer memory sizes. It should fit in the cache! The maximum temporary buffer size needed for the communication and reduction step of the counters is 147.31 KB, which is not critical and significantly smaller than the 64 GB of the not improved version.

The analysis and the example show the potential of the formulated improvement. In Fig. 7.4 and 7.5, the different approaches for decomposing the data are illustrated.

Until now, we have just looked how the number of counters increases in both methods depending on the number of iterations. However, we also have to care about the way the new buckets are created locally in the step 1 (*Reverse Sorting*). C3-Radix scans its n keys and rebuilds all buckets using the new radix. While building the buckets the counter array is updated, too. The scan can be done in $O(n)$, and an update of the buckets and counters can be done in $O(1)$. A practical drawback of this method is the following: As we know, the array of buckets and counters may get very big. Scanning the data means to access these arrays very irregularly. Hence, the memory hierarchy of the system is not used efficiently, and the time needed for step 1 will grow quickly beginning at a critical size of the radix.

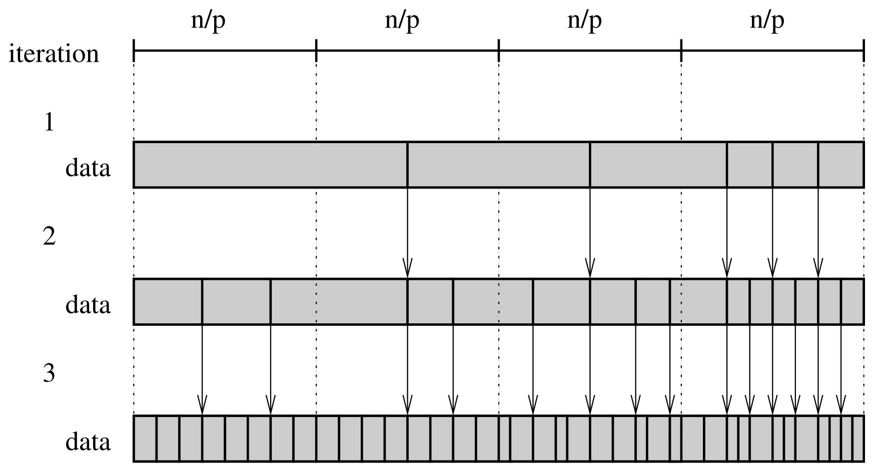


Figure 7.4: The figure illustrates the decomposition method of C3-Radix. The C3-Radix algorithm rebuilds every bucket in every iteration.

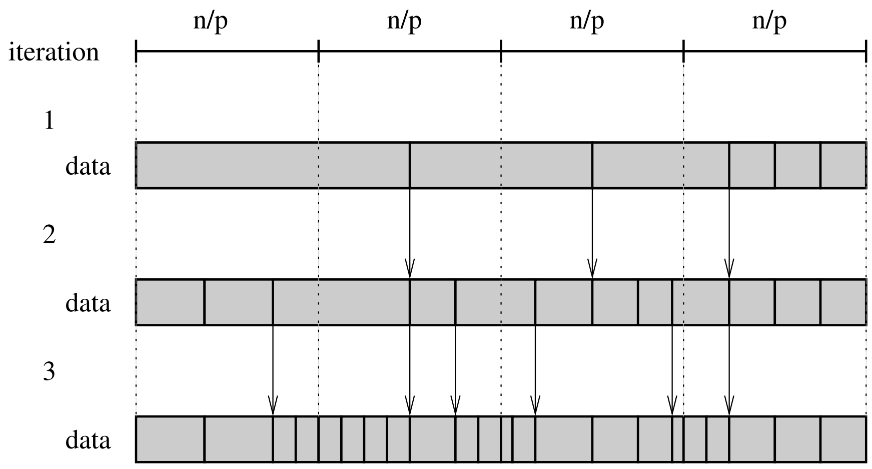


Figure 7.5: The figure illustrates the decomposition method of BCSP-Radix. BCSP-Radix only splits the buckets which may improve the load balance.

On the other side, the improved version does not always have to scan all local integer keys. It just scans the keys located in the buckets, which were decided to be rebuilt. However, in the worst-case all local keys are contained in such buckets. Further, these n keys are not stored in one array consecutively, therefore, there is an additional overhead for switching between the buckets. After scanning the buckets and building new buckets out of them, these buckets have to replace the old buckets. Assuming that all buckets are stored in an array, this operation can be done in $O(2^r)$, whereby r is the size of the initial radix in the first iteration. For small initial radix sizes n should be $\geq 2^r$, hence, the additional time needed is not a problem. For larger sizes step 1 is not the bottleneck of the algorithm as we saw in the example. Despite of that, we see that there might be situations for step 1, where C3-Radix is better than the improved version and vice versa, depending on the data distribution and the size of the radix.

In the following sections, we present experimental results that will show the behavior of the two algorithms for several data distributions. The improved algorithm is called balanced communication sensitive parallel radix sort (*BCSP-Radix*).

7.4 Experimental Tests

As in Chapter 5, our experimental tests were made on the Kepler-Cluster, [75]. Just to remember, this is a Linux-SMP cluster with two Pentium III processors (650 MHz) and 1 GB main memory per node. The nodes are connected by a Myrinet 1.28 GBit/s switched LAN. The whole cluster consists of 98 nodes.

As we showed in Section 7.3, it is sufficient to compare the first 3 steps of C3- and BCSP-Radix , because the steps 4 and 5 are the same as in the C3-Radix algorithm. Again, we implemented them with C++ and TPO++ [38], which is an object-oriented message-passing system build on MPI [57, 59].

Concerning the data distributions, we use two kinds of data sets. The first type are distributions already used in [70, 45, 47, 40] and, therefore, are called standard data distributions. The second type are data distributions, which leads to worst-case behavior. We will explain them in Section 7.4.2. Duplicates are allowed in all data sets. For all experiments, we try to achieve the best load balance possible. Since duplicates are allowed, a perfect load balance is not always possible. In general, we accept deviations of $\leq 1\%$ of n/p.

7.4.1 Standard Data Distributions

Our four standard data distributions are defined as follows, in which `MAX` is $(2^{31} - 1)$ for the integer keys, see also [70, 45, 47, 40].

1. **Random [R]**, the data set is produced by calling the C library random number generator `random()` consecutively. The function returns integer values between 0 and $2^{31} - 1$.

2. **Gaussian [G]**, an integer key is generated by calling the `random()` function 4 times, adding the return values and dividing the result by 4.

3. **Bucket Sorted [B]**, the generated integer keys are sorted into p buckets, obtained by setting the first n/p^2 keys at each processor to be random numbers in the range of 0 to $(\mathtt{MAX}/p - 1)$, the second n/p^2 keys in the range of $\mathtt{MAX}/p$ to $(2\mathtt{MAX}/p - 1)$, and so forth.

4. **Staggered [S]**, if the processor index i is $< p/2$, then all n/p integer keys at the processor are random numbers between $(2i + 1)\mathtt{MAX}/p$. Otherwise, all n/p keys are random numbers between $(i-p/2)\mathtt{MAX}/p$ and $((i - p/2 + 1)\mathtt{MAX}/p - 1)$.

The main problem of using radix sort is that we do not know how the data is distributed and, therefore, with which size of the radix we should start. Our aim is to minimize the total running time. Nevertheless, without knowing details about the data it is not possible to choose the size of the radix optimally. The algorithm should grant that the running time should be close or equal to its optimum independent of the radix size and data distribution . Hence, in our test, we varied the initial size of the radix from 5 to 12. We made experiments using 16M integer keys and 8, 16 and 32 processors. All tests produced similar results, therefore, we give the explanation of the results by means of the 16 processors test. The running times of this test is presented in Table 7.1.

BCSP-Radix									
radix	5	6	7	8	9	10	11	12	total mean
[R]	0.95|1	1.06|1	1.12|1	1.24|1	1.24|1	1.23|1	1.20|1	1.15|1	1.15
[G]	2.63|3	2.42|2	1.85|2	1.68|2	1.71|2	1.77|2	1.07|1	1.11|1	1.78
[B]	2.87|3	2.87|3	2.51|2	2.44|2	2.10|2	1.94|2	0.93|1	0.96|1	2.08
[S]	2.76|3	2.53|3	2.94|2	2.34|2	1.98|2	1.79|2	1.01|1	1.09|1	2.05
C3-Radix									
radix	5	6	7	8	9	10	11	12	total mean
[R]	0.96|1	1.05|1	1.16|1	1.23|1	1.21|1	1.19|1	1.16|1	1.14|1	1.14
[G]	2.87|3	1.63|2	1.92|2	2.83|2	4.65|2	10.74|2	1.06|1	1.12|1	3.35
[B]	2.47|3	4.45|3	1.67|2	2.09|2	3.74|2	10.02|2	0.93|1	0.96|1	3.29
[S]	2.49|3	4.23|3	1.75|2	2.05|2	3.62|2	9.62|2	0.99|1	1.08|1	3.23

Table 7.1: Test of the C3 and BSCP method on 16 processors (8 nodes with 2 processors) and 16M integer keys using 4 different data distributions, and varying the initial radix in a range from 5 to 12. The first number in each cell is the mean running time of all used processors for the first 3 steps. The second number is the number of iterations needed for the execution.

Interpretation of the Results

- Concerning the [R] distribution, both algorithms achieve similar running times for all sizes of the radix, because if only 1 iteration is necessary then they are the same. The best running time can be achieved by using the smallest radix, which is obvious, because then the data structures are small, too. Computation and communication benefit by that.

- In general, the overall best running time for all distributions can be achieved, if we use the smallest size of the radix, with which only 1 iteration is necessary. In our sample this is $r = 11$ for [G], [B] and [S]. Again, both algorithms are the same.

- If the radix is chosen < 11, then the situation is as follows. If the number of iterations needed to perform the algorithm is equal for several sizes of the initial radix, then C3-Radix is better if the size of the radix is minimal and BCSP-Radix is better for larger sizes of the radix. The reason why C3-Radix gets slower for larger sizes of the radix is founded in the increasing sizes of the bucket and counter arrays. Communication and computation gets much higher if the radix size increases dramatically due to further iterations. BCSP-Radix is much more stable, because the more iterations are necessary, the smaller the buckets are that have to be rebuilt. Furthermore, the size of the counter array for the communication is bounded by $O(2^r)$ as we saw in Section 7.3. Therefore, the communication cost is stable, too. If BCSP-Radix is worse, this is because of the running time of step 1. In these cases, due to the small radix, the data is distributed in a small number of buckets ($\leq p-1$). Hence, each bucket has to be rebuilt and each processor has to scan all integer keys stored locally. The additional time needed for replacing all buckets by 2^r new buckets leads to the worse behavior. This situation is illustrated in Fig. 7.6 for radix 6. The figure is based on the results presented in Table 7.1. We compare the composition of the total running time for a sequence of radix sizes, which lead to the same number of iterations. As predicted,the running time for C3-Radix grows exponentially with the size of the radix, while the running time for BCSP-Radix is very stable and increases very slowly.

- An essential observation can be made looking at the mean of the processor mean times (see column *total mean* in Table 7.1). BCSP-Radix is better for [S], [B], [G] and of course equal for [R]. While C3-Radix has some runaways in all distributions, BCSP-Radix is much more stable. This is important, because normally we do not know the behavior of the data with respect to the chosen radix. By using BCSP-Radix, it is guaranteed that the first three steps do not destroy the total running time of the whole algorithm.

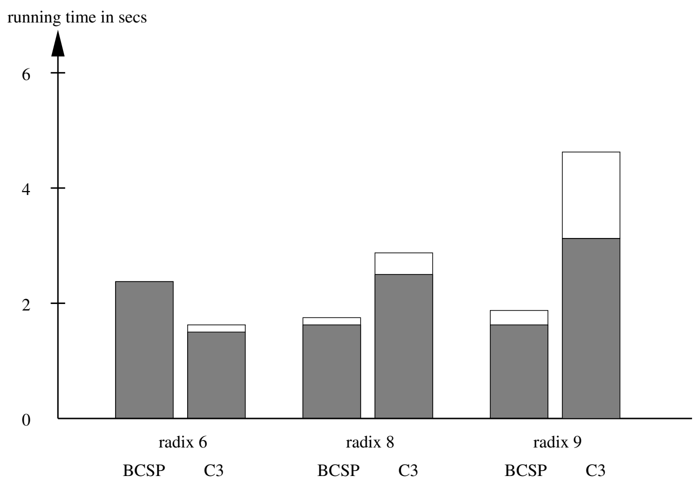


Figure 7.6: Comparison of the running times of C3-Radix and BCSP-Radix using 16M integer keys, 16 processors, and data distribution [G]. The shaded part of the running time represents step 1, the white part represents step 2. The times for 3 different sizes of the initial radix (6, 8, and 9) are considered to show the development of the running time.In all situations 2 iterations are necessary.

7.4.2 Worst-Case Data Distributions

For the BCSP-Radix algorithm, the worst-case data-distribution can be described as follows. After the first iteration, the data is partitioned into $p-1$ buckets of equal size and the other buckets are empty. The algorithm decides to rebuild all $p-1$ buckets with a larger radix. However, the data is chosen in the way that the next bits are the same for all keys until the last r bits begin. That means that in each iteration the $p-1$ buckets remain unchanged until the last iteration. This is the worst case, because BCSP-Radix has to scan all keys in each iteration, and $p-1$ is the maximum number of buckets, which achieve this situation. Using this data set with C3-Radix does also lead to the maximum number of iterations possible. This data set is not constructed very artificially. The keys are uniformly distributed within a small range of bits and duplicates are allowed. Hence, this might also occur in a well-distributed environment. As we know from the example in Section 7.3, we cannot perform all iterations for 32 bit keys with the C3-Radix, because the main memory of our SMP nodes is limited to 1 GB (using only 1 processor per node!). Therefore, the data set is constructed in the way that the nodes of the cluster will not collapse (≤ 20 bits). However, we will show the behavior of BCSP-Radix performing all possible iterations.

Interpretation of the Results

Table 7.2 presents an example of the comparison between the algorithms, where the data is distributed in the way that both algorithms have to perform 4 iterations, which means that the data can be partitioned looking at the first 20 bits. BCSP-Radix is much better than C3-Radix. The difference between both would even grow if further iterations were necessary. Unfortunately, C3-Radix cannot iterate further without aborting due to memory limitations.

Algorithm	Step 1	Step 2	Step 3	$\sum$
C3	4.79	5.88	0.15	10.82
BCSP	5.35	0.02	< 0.01	5.37

Table 7.2: Test with 16 processors, 16M integer keys, and an initial radix of 5. The numbers are the mean times (in sec.) of the processors for a worst-case situation where 4 iterations are necessary.

In Table 7.3, the running times for BCSP-Radix making 6 iterations (30 bits) are illustrated. We have to adhere that in this case BCSP-Radix is even faster than C3-Radix performing only 4 iterations. Although this is a worst-case situation for BCSP-Radix, the running time is better than the worst time achieved with C3-Radix for [S], [B] and [G] and radix 10 where 2 iterations were necessary (see Table 7.1).

Algorithm	Step 1	Step 2	Step 3	$\sum$
BCSP	7.72	0.04	< 0.01	7.76

Table 7.3: Test with 16 processors, 16M integer keys and an initial radix of 5. The numbers are the mean times (in sec.) of the processors for the worst case situation where 6 iterations are necessary. Only BCSP-Radix is able to terminate its execution.

7.5 Summary

In this chapter, we illustrated the development path from a sequential algorithm to a hierarchical sensitive parallel algorithm for Radix sort. We showed drawbacks and improvements for all parallel versions. Besides the improvements concerning load balancing, the most important improvement considers the reduction of communication cost. This improvement was motivated by distributed parallel architectures like SMP clusters, because here the practical drawback is much higher than in shared-memory machines.

Hence, the development path illustrates the method of *hierarchical sensitive design*, because communication is avoided as much as possible and in contrast local computation is preferred.

Further, in the previous sections, we suggested improvements with respect to the data decomposition for the currently fastest parallel radix sort algorithm. Instead of rebuilding all buckets for each iteration with a larger radix size, BCSP-Radix only rebuilds those buckets, which help to find a better load balance for the proceeding steps of the algorithm. Due to this optimization, the computation, communication and memory requirements are much better. In experimental tests, we showed that the average general behavior of BCSP-Radix for standard data distributions is superior. While C3-Radix has several situations where the execution time deverges extremely from the mean time, BCSP-Radix behaves much more stable and predictable. For worst-case data distributions, the advantage of using BCSP-Radix is even bigger. While the non-optimized algorithm may not terminate due to memory constraints, BCSP-Radix is even in the position to achieve reasonable running times. The memory requirements per processor do not violate the capacity of custom workstations.

Chapter 8

Summary, Conclusions and Outlook

In Chapter 1, we introduced the *design chain* for efficient parallel applications. An efficient algorithm for a certain parallel architecture is developed and analyzed best on a cost model that reflects the key characteristics of the target machine. Further, an appropriate programming model is necessary that fits to the target machine as well and with which the algorithm can be implemented directly. The question is posed how the situation for an emerging architecture called SMP cluster is. This kind of parallel computer is characterized by a parallel hierarchy. On the one hand side, we have parallelism within the nodes, where multiple processors have a shared-memory. On the other hand, parallelism between the nodes uses the interconnection network. Generally, communication over shared-memory is faster than over communication networks. Hence, we have also a communication hierarchy. Additionally, even the network may consist of several levels with different time behaviors. These levels can appear due to multiple switches, Grid- or Metacomputing technologies.

Hence, in Chapter 2, we gave an overview on parallel architectures, cost models and programming models in order to verify if there is a rational chain for SMP clusters. Besides an introduction to parallel architectures, the chapter explained that there is a trend to hierarchical structures , especially to SMP clusters. We gave arguments why this trend started and why it will last for several years.

Concerning cost models, we introduced the concept of parallel bridging models that postulates the aim of building a general-purpose cost model on which efficient algorithms can be design for all parallel computers independently of their system-specific features. Despite of this generality, the algorithms should also perform well in practice. We described the most popular parallel bridging models namely BSP, LogP and QSM. Additionally, we reviewed extensions of these models that try to respect hierarchical memory and machine structures. Examples for such extensions are BSP* and the D-BSP model.

Further, we described the two main programming models for distributed and shared-memory architectures. For distributed-memory machines, the message-passing model is used. Each processor has its local memory. If data has to be exchanged between processors, this is done by messages. The most accepted library for message-passing is MPI (message-passing interface). Shared-memory programming is usually done by threads. The most accepted library is OpenMP with which it is possible to define parallel regions in a sequential code. These regions are executed according to work sharing directives by a predefined number of threads. After the parallel region, the threads join again and only a master thread continues.

A new approach for programming SMP clusters is the hybrid-programming model that proposes to use message-passing between the nodes and threads within the nodes of an SMP cluster. This model has the highest potential to be very efficient for SMP clusters, because it uses the most adequate model for the corresponding parts of the SMP cluster. A drawback is the increased complexity, which leads to problems for program maintenance, profiling and debugging.

The chapter concludes with the fact, that there is no obvious design chain for parallel algorithms for SMP clusters. There is nearly no work which tries to optimize algorithms for the platform. This conclusion was taken as a motivation to work out a design chain and to develop methods, with which it is possible to design efficient parallel algorithms for SMP clusters. The programming model with the most potential on SMP clusters is the hybrid-programming model. Hence, our idea was to develop the missing element for the chain which is a cost model that reflects the programming model and the architecture of SMP clusters.

In Chapter 3, we build a cost model for the design and analysis of parallel algorithms for SMP Clusters. The model is called κNUMA. The model is based on elements of several accepted parallel bridging models, especially on the BSP model and extended by the characteristics of SMP clusters and the hybrid-programming model. Hence, the model introduces different cost measures for inner- and inter-node communication. We presented the usage of the model by analyzing broadcast problems, and showed that even for this basic communication operation, algorithms that were suggested by the BSP model are too coarse. With the help of the κNUMA model, it was possible to develop a more accurate algorithm for the problem.

The chapter concludes that it makes sense to respect the hierarchical structure of SMP clusters within the design process, because otherwise the resulting algorithms may become sub-optimal.

One problem with the κNUMA model is that the analysis gets very complicated even for simple algorithms like the (personalized) broadcast. On the other hand, the κNUMA model can be regarded as a super-model. It is not always necessary to use the full model, because either there is a certain architecture that reduces the set of parameters, or the analyzed problem itself does not need a complete model. In further chapters, we showed cases where reduced instances of the κNUMA model can be used and make the analysis more feasible but not less significant. Another problem is that the theoretical results were not verified in practice. In addition, this was done for other problems in the next chapters.

In Chapter 4, we reviewed and presented methods for designing and optimizing parallel algorithms for SMP clusters. The overall design process consists of several stages, beginning at the problem definition (or sequential algorithm), followed by a parallel algorithm, followed by a parallel SMP cluster algorithm and ending in an optimized parallel SMP cluster algorithm. Each method has its individual start and end stage within the design process. Hence, it is possible to design an algorithm by using a chain of methods. In each of the following chapters, we presented case studies for developing efficient algorithms for several problems by applying a certain chain of methods.

In Chapter 5, we tried to reduce communication costs by storing data redundantly. On the other hand, we showed that due to certain a transfer method the storage of unnecessary redundant data could be avoided. Therefore, we called this method *Exploitation of Data Redundancy*. The method was explained using the problem of dense matrix-vector-multiplication. The analysis is applied on the κNUMA model with $\kappa = 1$. This reflects the characteristics of a special SMP cluster, on which we made experimental tests to verify the model's predictions. We formulated a general parallel algorithm for the dense matrix-vector multiplication and examined several data-distributions. Depending on the different matrix distributions, there has to be stored more or less elements of the vector with which the matrix has to be multiplied. For all distributions, the same amount of matrix elements has to be stored. Using different data distributions leads to two extremes

1. A minimum number of elements of the vector has to be stored for each processor, but the algorithm needs an all-to-all communication pattern. For large-scale applications, it might be more important to save memory than time for communication, because memory might be a critical resource.

2. No communication costs are necessary by storing more elements of the vector per processor. If there is enough memory space, then this is obviously the best solution.

Further, we showed in this chapter that a hierarchical analysis with the κNUMA model is superior to a non-hierarchic analysis by giving an example where an efficient distribution for the basic algorithm suggested by the BSP model can lead to an inefficient implementation on SMP clusters. We showed that this is avoided even by using a reduced κNUMA model with $\kappa = 1$.

The second case study attempts to improve performance by the *Adaptation of Communication Patterns* to hierarchical structures of SMP clusters and was presented in Chapter 6. This was explained at the problem of transposing a matrix that is the result of a matrix-multiplication in a parallel environment. While in the sequential case, the transposition can be done by just exchanging the indices of the columns and rows, in parallel computation the matrix is evenly distributed among the processors. Depending on this distribution, a certain communication pattern has to be executed by the processors. For the block-cyclic distribution, this often leads to an all-to-all communication operation between the processors.

By using a special data distribution this communication step can be avoided. The main idea is to multiply the matrices in the way that afterwards all blocks that have to be exchanged by a transposition operation were computed by processors that reside in the same SMP node of the cluster. In this case, no communication is necessary, because the blocks are already in the same memory and because of the hybrid model even in the same memory space. The transposition can be done by local computation.

The last method presented in the thesis is called *Hierarchical Sensitive Design*, and was explained in Chapter 7. In general, algorithms that try to minimize communication and try to shift the complexity to local computations have great relevance for hierarchical architectures and are called *hierarchical sensitive*. Especially algorithms that try to decompose and distribute the data in a pre-processing step and continue with local computations belong to this class. Communication only takes place in this step, afterwards only local computation is necessary to produce the results. As an example for the development of such algorithms, we considered the sorting method called *Radix Sort*. We presented various algorithms from the sequential to the latest parallel algorithm, and we described the motivations for each respective improvement.

Briefly, the classical parallel version of Radix sort, sorts the integers by scanning and comparing their binary representation from the least significant to the most significant bit. This always leads to all-to-all communication patterns. This can be avoided by starting the sorting the other way round. The most efficient algorithm is based on two parts. The first part divides the set of integers in p equal sized sets, where p is the number of processors. How this can be done is the core of the hierarchical sensitive algorithm. Finally, these sets are sorted locally.

In parallel computation, a lot of effort is done in accelerating applications by technical improvements like optimizing libraries or compilers. One aim of this work was to show that it is not enough to trust in the efficiency of the libraries of the used programming model and to show that further improvements due to algorithmic optimizations are possible. Of course, it is also possible to achieve a good performance by just using the programming libraries without considering the hierarchical structures. However, treating modern parallel architectures as non-hierarchic, will in general not lead to the best performance possible. The situation is comparable to that of memory hierarchies within a computer, where the consideration of data locality does also lead to remarkable performance improvements. Hence, it is also required to improve algorithms to hierarchical SMP clusters. Algorithmic optimizations can be regarded as additional improvements in the sense that just optimized algorithm are capable to exploit the technical improvements best.

This thesis shows methods and case studies how program developers can try to develop and transform their algorithms in the way that they consider hierarchical structures. The methods can be used to develop parallel algorithms from scratch, to transform existing parallel algorithms to SMP clusters and to optimize them for SMP clusters concerning communication costs. The resulting algorithms can be analyzed with the help of the κNUMA cost model that predicts their practical performance.

Although, the formulation of a cost model seems to be too theoretical for the practical daily use, it helps to understand parallel hierarchy and helps to produce ideas how algorithms may be adapted or changed best in order to run efficiently on SMP clusters. The idea to have one single model for all parallel computers on which developers can analyze and design algorithms that perform well in practice is very interesting, but should not be taken too literally. As we showed, in practice real machines have certain very special characteristics, which influence the performance of programs dramatically. In general, these characteristics are not considered by parallel bridging models, so it seems to be a better solution to extend these models with the special characteristics of the underlying machine and to show how algorithms can be transformed best. Of course, this comprises the risk of producing too complex models. However, as we showed, it makes sense to regard these models as super-models that can be reasonably reduced e.g. by fixing parameters for a certain computer. This makes the analysis more feasible, without loosing prediction quality. The approach of extending parallel bridging models with special characteristics of the underlying machine is not limited to SMP clusters. We considered them as an example, because they have great relevance in the supercomputing area.

Anyhow, this approach for developing applications is very low level. In the previous chapters, we gave optimizations for basic operations and algorithms. Normally, a programmer should have the possibility to use these operations and algorithms as black-boxes. Of course, there are libraries that provide similar functionality for many architectures and also for SMP clusters. In general, they are simply “optimized” by recompiling on the target platform. Hence, it would be desirable to have libraries that are optimized algorithmically and technically. This combination should produce the best performance (e.g. an optimized algorithm library based on an optimized MPI library).

Hence, we propose two research paths. The first path is concerned with the development of better algorithms as we tried to show in this thesis. The second path deals with compilers, programming models and software tools in general with which it is possible to make efficient implementations for SMP clusters. For example, the hybrid-programming model has the best potential but it is still too complex to create, maintain and debug programs based on it. The synthesis of both paths will lead to a programming environment comparable with that of sequential computing. Only this way will make it possible for parallel computation to make the step from the scientific domain into the business domain.

In addition, business processes contain very time consuming tasks that may profit from their parallelization. Cheap parallel machines like SMP clusters, an efficient standardized programming environment and a huge amount of efficient libraries are arguments for companies that suffer from slow sequential implementations to do investments. This step will bring parallel computation into new dimension.

Appendix A

Related Publications

The chapters of this thesis are founded on our refereed publications in international conferences, journals and books of the last years. In the following, we give an overview which chapter is based on which publication, and where the publication was done.

A.1 Conferences

- *Martin Schmollinger, Michael Kaufmann. "κNUMA: A Model for Clusters of SMP-Machines". In Proceedings of the 4th Conference on Parallel Processing and Applied Mathematics (PPAM 2001), Naleczow, Polen, September 2001. LNCS 2328, Springer Verlag.*

  ⟶ Chapter 3, 4.

- *Martin Schmollinger, Michael Kaufmann. "Algorithms for SMP Clusters. Matrix-Vector Multiplication". Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS), Ft. Lauderdale, Florida, USA, April 2002. IEEE Computer Society.*

  ⟶ Chapter 5.

- *Martin Schmollinger, Michael Kaufmann. "Matrix Transpose on SMP Clusters", 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS), MIT, Camebridge, USA. November 2002, Accepted for publication.*

  ⟶ Chapter 6.

- *Martin Schmollinger. "Improving Communication Sensitive Parallel Radix Sort for Unbalanced Data", Proceedings of the 9th International Euro-Par Conference, Klagenfurt, Austria. August 2003. LNCS 2790, Springer-Verlag.*

  $\longrightarrow$ Chapter 7.

A.2 Journals

- *Martin Schmollinger, Michael Kaufmann. "Designing Parallel Algorithms for Hierarchical SMP Clusters". In International Journal of Foundations of Computer Science special issue on advances in parallel and distributed computational models, 14(1):59-78, World Scientific Publishing Company. February 2003.*

  $\longrightarrow$ Chapter 3, 4 and 5.

A.3 Book Chapters

- *Massimo Coppola, Martin Schmollinger. Chapter "Hierarchical Models and Software Tools for Parallel Programming". in "Algorithms for Memory Hierarchies", Advanced Lectures, LNCS 2625, Springer Verlag, 2003.*

  $\longrightarrow$ Chapter 2

Bibliography

[1] V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. J. K. Tan, editors. *Computational Science - ICCS 2001, Part II*, volume 2074 of *LNCS*, 2001. 137, 138, 140, 142

[2] 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. 38

[3] F. M. auf der Heide and R. Wanka. Parallel bridging models and their impact on algorithm design. In Alexandrov et al. [1], pages 628–637. 22, 27, 29

[4] S. Baden and S. Fink. A Programming Methodology for Dual-Tier Multicomputers. *IEEE Transactions on Software Engineering*, 26(3):212–226, 2000. 45

[5] D. 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. 43

[6] 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. http://hpc.eece.unm.edu/papers/3798.ps. 73, 114

[7] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 94035-1000, December 1995. 36

[8] G. Bell and J. Gray. High Performance Computing: Crays, Clusters, and Centers. What Next? Technical Report MSR-TR-2001-76, Microsoft Research, Microsoft Corporation, August 2001. 16, 19

[9] S. Benkner. VFC:The Vienna Fortran Compiler. *Scientific Programming*, 7(1):67–81, 1999. 44

[10] S. Benkner and V. Sipkova. Language and Compiler Support for Hybrid-Parallel Programming on SMP Clusters. In *Proceedings of the 4th International Symposium on High Performance Computing*, volume 2327 of *LNCS*. Springer, 2002. 44

[11] M. 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 *LNCS*, pages 349–359, 1999. 29

[12] 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. [1], pages 579–588. 29, 59

[13] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, , and R. C. Whaley. ScaLAPACK: A Linear Algebra Library for Message-Passing Computers. In *Proceedings of the SIAM Conference on Parallel Processing*, 1997. 97

[14] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine. In *Proceedings of Sysmposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991. 117

[15] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library - Design, Implementation and Performance. In *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, April 1999. 27

[16] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In *Proceedings of the Supercomputing Conference*. IEEE/ACM, 2000. 40, 41, 73

[17] F. Cappello and O. Richard. Performance Characteristics of a Network of Commodity Multiprocessors for the NAS Benchmarks Using a Hybrid Memory Model. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques PACT*. IEEE/IFIP, October 1999. 41

[18] F. Cappello, O. Richard, and D. Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *Proceeding of the 6th High Performance Computer Architecture Conference*. IEEE, January 2000. 41

[19] A. A. Chien. Computing platforms. In Foster and Kesselman [32], chapter 17. 17

[20] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA:Parallel Universal Matrix Multiplication Algorithms on Distributed Concurrent Computers. *Concurrency: Practice and Experience*, 6:543–570, 1994. 96

[21] J. Choi, J. J. Dongarra, and D. W. Walker. Parallel matrix transpose algorithms on distributed memory concurrent computers. *Parallel Computing*, 21(9):1387–1405, 1995. 97

[22] 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. 41

[23] M. Coppola and M. Schmollinger. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS Advanced Lectures*, chapter Hierarchical Models and Software Tools for Parallel Programming. Springer-Verlag, 2003.

[24] 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. *Communications of the ACM*, 39(11):78–85, November 1996. 25

[25] 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, Puerto Rico, 1999. 25

[26] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory of Computing Systems*, pages 1–31, Sept. 2002. Electronic version of the journal, DOI: 10.1007/s00224-002-1066-2. 25

[27] E. Demaine. A Threads-Only MPI Implementation for the Development of Parallel Programs. In *Proceedings of the 11th International Symposium on High Performance Computing Systems, HPCS*, pages 153–163, 1997. 37

[28] J. Dongarra and V. Eijkhout. Numerical linear algebra algorithms and software. *CAM (Numerical) Linear Algebra*, 31(4), 1999. 95

[29] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972. 11

[30] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th anual Symposium on Theory of Computing*, pages 114–118. ACM, 1978. 20

[31] I. Foster. *Designing and Building Parallel Programs, Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Publishing Company, 1994. 9, 67

[32] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Pub., July 1998. 17, 19, 138

[33] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic Sorting and Randomized Median Finding on the BSP Model. In *Proceedings oth the 8th ACM Symposium on Parallel Algorithms and Architectures*, pages pp.223–232, 1996. 117

[34] P. B. Gibbons, Y. Mathias, and V. 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. 26, 55

[35] A. Grama, V. Kumar, S. Ranka, and V. Singh. Architecture independent analysis of parallel programs. In Alexandrov et al. [1], pages 599–608. 20

[36] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996. 36

[37] W. D. Gropp and E. Lusk. *User's Guide for* `MPICH`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6. 36

[38] T. Grundmann, M. Ritt, and W. Rosenstiel. Object-Oriented Message-Passing with TPO++. In *EURO-PAR 2000, Parallel Processing*, volume 1900 of *LNCS*, pages 1081–1084. Springer Verlag, 2000. 32, 91, 123

[39] O. Haan. Matrix Transpose with Hybrid OpenMP/MPI Parallelization. In *Second meeting of IBM SP Scientific Computing User Group* http://www.spscicomp.org/2000/userpres.html#haan, 2000. 97

[40] D. R. Helman, D. A. Bader, and J. JáJá. Parallel Algorithms for Personalized Communication and Sorting With Experimental Study. In *Proceedings of the IEEE Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 211–220, 1996. 123

[41] D. R. Helman and J. JáJá. Sorting on Clusters of SMPs. *Informatica: An International Journal of Computing and Informatics*, 23, 1999. 73, 117

[42] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP Programming Library. Technical Report PRG-TR-29-97, Oxford University Computing Laboratory, 1997. 27

[43] Y. Hu, H. Lu, A. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. In *Proceedings of the 2nd Merged Symposium International Parallel*

and Distributed Symposium/Symposium on Parallel and Distributed Processing (IPPS/SPDP). IEEE, 1999. 38

[44] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. 70, 83

[45] D. Jiminez-Gonzales, J. Larriba-Pey, and J. Navarro. Communication Conscious Radix Sort. In *Proceedings of the International Conference on Supercomputing*, pages 76–82. ACM, 1999. 117, 119, 123

[46] D. Jiminez-Gonzales, J. Larriba-Pey, and J. Navarro. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS, Advanced Lectures*, chapter Case Study: Memory Conscious Parallel Sorting, pages 358–378. Springer Verlag, 2003 to appear. 114

[47] D. Jiminez-Gonzales, J. Navarro, , and J. Larriba-Pey. Fast Parallel In-Memory 64 Bit Sorting. In *Proceedings of the International Conference on Supercomputing*, pages 114–122. ACM, 2001. 117, 123

[48] W. 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. 18

[49] B. Juurlink and H. Wijshoff. *EURO-PAR 96, Parallel Processing*, volume 1124 of *LNCS*, chapter The E-BSP Model: Incorporating Unbalanced Communication and General Locality into the BSP Model, pages 339–347. Springer, August 1996. 59

[50] D. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973. 111, 113

[51] M. Kowarschik and C. Weiß. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS, Advanced Lectures*, chapter An Overview of Cache Optimzation Techniques and Cache-Aware Numerical Algorithms, pages 213–232. Springer Verlag, 2003. 96

[52] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings Publ. Company, 1994. 12, 13, 14, 97

[53] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992. 13

[54] W. F. McColl. *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, chapter Scalable Computing, pages 46–61. Springer Verlag, 1995. 83, 90

[55] J. Merlin, D. Miles, and V. Schuster. The Portland Group Distributed OMP: Extensions to OpenMP for SMP-Cluster. In *Proceedings of 2nd European Workshop on OpenMP (EWOMP)*, Edinburgh, UK, September 2000. Edinburgh Parallel Computing Centre. Electronic Proceedings, `http://www.epcc.ed.ac.uk/ewomp2000/`. 39

[56] 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. 30

[57] 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. 123

[58] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. 30

[59] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. 123

[60] H. Meurer, E. Strohmaier, J. Dongarra, and H. Simon. TOP500 Supercomputer Sites, `http://www.top500.org`, June 2003. 16, 82

[61] OpenMP Forum. OpenMP C and C++ Application Program Interface, Version 2.0. `http://www.openmp.org`, March 2002. 33, 34, 36, 83

[62] V. Ramachandran. Parallel algorithm design with coarse-grained synchronization. In Alexandrov et al. [1], pages 619–627. 26

[63] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proceedings of the 1st European Workshop on OpenMP (EWOMP)*, pages 32–39. Lund University, Lund, Sweden, 1999. Electronic proceedings, `http://www.it.lth.se/ewomp99/`. 38

[64] M. Schmollinger. Improving Communication Sensitive Parallel Radix Sort for Unbalanced Data. In *Proceedings of the 9th International Conference Euro-Par*, number 2790 in LNCS. Springer Verlag, August 2003.

[65] M. Schmollinger and M. Kaufmann. A Model for Clusters of SMP-Machines. In *Conference on Parallel Processing and Applied Mathematics*, volume 2328 of *LNCS*, pages pp. 42–50. Springer Verlag, 2001.

[66] M. Schmollinger and M. Kaufmann. Algorithms for SMP Clusters, Dense Matrix-Vector Multiplication. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, April 2002.

[67] M. Schmollinger and M. Kaufmann. Designing Algorithms for Hierarchical SMP Clusters. *International Journal of Foundations of Computer Science*, 14(1), February 2003.

[68] R. Sedgewick. *Algorithms*. Addison-Wesley, 1992. 111, 113

[69] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992. 117

[70] A. Sohn and Y. Kodama. Load Balanced Parallel Radix Sort. In *Proceedings of the International Conference on Supercomputing*, pages 305–312. ACM, 1998. 114, 123

[71] T. Takahashi, F. O'Carroll, H. Tezuka, A. Hori, S. Sumimoto, H. Harada, Y. Ishikawa, and P. Beckman. Implementation and Evaluation of MPI on an SMP Cluster. In *Proceedings of the International Parallel and Distributed Symposium/Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, pages 1178–1192. IEEE, 1999. 36

[72] R. Thakur, W. Gropp, and E. Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Proc. of SC98: High Performance Networking and Computing*. IEEE, Nov. 1998. 32

[73] P. D. L. 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 *LNCS*, pages 352–360, 1996. 28

[74] T. Ungerer. *Parallel Rechner und Parallele Programmierung*. Spektrum Akademischer Verlag, 1997. 5

[75] University of Tübingen (SFB-382). http://kepler.sfb382-zdv.uni-tuebingen.de/. 81, 91, 123

[76] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990. 9, 21, 53, 63

[77] A. van der Steen and J. Dongarra. Overview of Recent Supercomputers. `http://www.top500.org/ORSC/2002/`, July 2002. 11, 14

[78] J. S. Vitter. External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001. 20

Index