Dissertation

Versatility of Bulk Synchronous Parallel Computing: From the Heterogeneous Cluster to the System on Chip

Olaf Bonorden

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik und Heinz Nixdorf Institut

February 2008

Zusammenfassung

Der Bedarf an Rechenleistung in den Wissenschaften und der Industrie steigt ständig an. Bei der Entwicklung schnellerer Prozessoren gelangt man allerdings an physikalische Grenzen, so dass zur weiteren Leistungssteigerung sehr viele Prozessoren zusammen als Parallelcomputer benutzt werden müssen. Zur Nutzung dieser Supercomputer ist es wichtig, allgemeine Modelle und Werkzeuge zu haben, die es erlauben, unabhängig von der speziellen Hardware effiziente Algorithmen zu entwickeln und zu implementieren.

In dieser Dissertation werden Modelle für parallele Systeme vorgestellt, ein Überblick über Algorithmen für diese Modelle gegeben und effiziente Implementierungen für verschiedene Architekturen entwickelt. Der Schwerpunkt liegt dabei auf der Familie der *Bulk Synchronous Parallel* Modelle, da diese die Entwicklung portabler, aber trotzdem effizienter paralleler Programme erlauben.

Für die Implementierungen werden zwei Architekturen betrachtet: ein On-Chip-Parallelcomputer und Workstation-Cluster. Im Bereich des On-Chip-Systems zeigt die Arbeit, wie das benutzte Modell die Entwicklung applikationsunabhängiger, effizienter, paralleler Systeme unterstützen kann.

Die Workstation-Cluster, auf denen nur freie Rechenkapazitäten genutzt werden dürfen, stehen auf der anderen Seite des Spektrums paralleler Systeme. Sie unterscheiden sich vom On-Chip-System nicht nur durch viel größere Latenzen, geringere Kommunikationsbandbreite und größeren Arbeitsspeicher, sie können auch heterogen sein, d. h., verschiedene Computertypen enthalten. Hierdurch und durch die variable, sich ständig ändernde, nutzbare Rechenkapazität der einzelnen Knoten ergeben sich besondere Herausforderungen, z. B. Lastbalancierung. Hierfür stellt die Arbeit eine Implementierung vor, welche mittels virtueller Prozessoren und deren Migration die Last gleichmäßig im Netzwerk verteilt.

Exemplarische Implementierungen zeigen, dass die Idee eines allgemeinen Modells funktioniert, d. h., dass ein Algorithmus für dieses Modell, sogar ein Programmquelltext, zu effizienten Implementierungen auf unterschiedlichen Systemen führen kann.

Reviewers: Prof. Dr. Friedhelm Meyer auf der Heide, University of Paderborn
Prof. Dr.-Ing. Ulrich Rückert, University of Paderborn

Acknowledgements

First and foremost, I wish to thank my advisor, Prof. Dr. Friedhelm Meyer auf der Heide, for his great support over all the years. I had a pretty good time and a very nice atmosphere in his research group Algorithms and Complexity. I always had the freedom to choose the topics and direction of my research.

Furthermore, I would like to thank all (former) members of the research group Algorithms and Complexity and all participants of the projects DFG-Sonderforschungsbereich 376 and GigaNetIC, especially the persons with whom I closely collaborated in research, teaching, or just in daily conversations. I am grateful to Ingo Rieping, who introduced me to bulk synchronous parallel computing.

For the technical support, my thanks go to the members of the IRB, especially Ulrich Ahlers and Heinz-Georg Wassing, who always tried to satisfy my needs concerning hardware and software, and to the Paderborn Center for Parallel Computing (PC^2), namely Axel Keller, who always reanimated nodes of the system killed by my experiments.

Special thanks I owe to Bastian Degener and Marc Rautenhaus, who proof-read parts of this thesis.

Last but not least, I would like to thank Stefanie for her patience, empathy, love, and all the other things that make it so worthwhile to know her. She turned even the most stressful days into a wonderful time, nowadays assisted by Emely's and Nele's enthusiastic welcome each day after work.

Olaf Bonorden

To those who cannot read these lines anymore:

Heide Bonorden

Contents

Chapter 1

Introduction

1.1 High Performance Computing

High performance computing has become an ubiquitous component of scientific work in natural and engineering sciences. Due to the well-known physical limitations of increasing the computational power of individual processing elements, parallel systems have become particularly important in recent years.

1.1.1 Technological Progress

Processing elements are becoming increasingly faster. *Moore's Law* [Moo65, Moo98] is well-known for its statement that the complexity of microprocessors doubles every two years, a time interval nowadays adapted to 18 months. In the early days of processor design performance of a system could easily be improved by either increasing the clock frequency or by using more transistors. Since 1971 the number of transistors of the central processing unit (CPU[2]) has been increased from 2,250 to more than 299 millions in 2006 (Table 1.1). During the same time, arithmetic units were enlarged from 8 bits to 64 bits word size and support for the direct execution of increasingly complex instructions (e. g., multiplication) and floating point numbers was added. Later, sophisticated techniques were introduced to

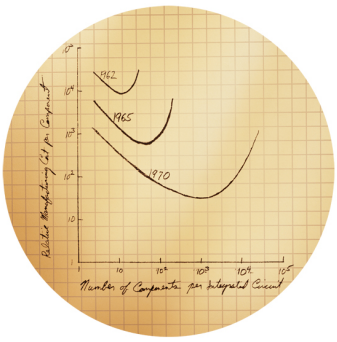


Figure 1.1: Moore's Law

[1] "In 1965, Gordon Moore sketched out his prediction of the pace of silicon technology. Decades later, Moore's Law remains true, driven largely by Intel's unparalleled silicon expertise."[Int06]

[2] In this thesis we will denote CPU by processor because we are not dealing with other processing units like e. g. graphic processing units (GPU).

Table 1.1: Die Sizes

Year	Processor	Transistors
1971	Intel 4004	2,250
1978	Intel 8086	29,000
1979	Motorola 68000	68,000
1989	Intel 80486	1,180,000
2001	IBM-Motorola Power4	174,000,000
2002	Intel Itanium II (McKinley)	410,000,000
2005	AMD Athlon 64	200,000,000
2006	Intel Pentium Xeon	291,000,000
2007	Intel Itanium II (Montecito)	1,720,000,000

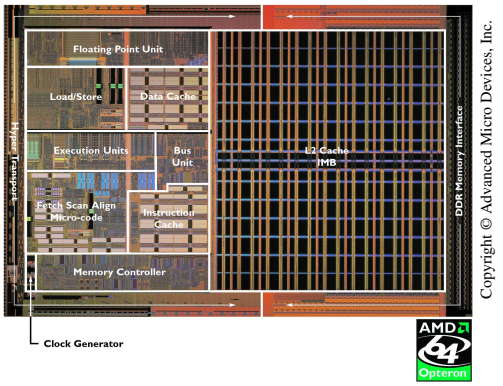


Figure 1.2: AMD Opteron

further increase the clock speed and to hide latencies. Examples are out-of-order execution, branch prediction, or speculative execution.

Another approach to utilize transistors is caching. Indeed, most transistors of modern processors are used for caches, for instance, the Intel Itanium II *Montecito* uses $1.55 \cdot 10^9$ of its $1.72 \cdot 10^9$ transistors for caches (24 MB). Figure 1.2 shows the die of an Opteron processor, although it has only 1 MB cache, the caches occupy more than half of the chip area.

1.1.2 Parallelism

The demand of computational power grows even faster than the increase in processor speed. To accommodate this demand, many processing units are combined to build large parallel machines. Parallel computers have been on the market for more than 20 years now, and much experience has been gained in designing such systems. Various design ideas have leed to a wide range of parallel architectures, which differ in the type of the processing unit (standard CPUs, vector processors, et cetera) and use quite different communication structures like shared memory or various network topologies.

Although parallel systems are widely used in high performance computing, for instance, for complex simulations in science and engineering (e. g., fluid mechanics or crash tests). Modern office computers often have multicore processors, i. e., they have two or more processor cores inside the CPU. They are thus parallel computers, but the number of processing elements is very small and most standard software does not utilize the parallelism.

Software is the major reason why massive parallel systems are not widely used, although such systems could be made available at low cost. In the beginning of parallel computing, most algorithms were developed and analyzed for specific parallel computers, for example, sorting algorithms for computers with a hypercube network topology

[CP93]. When a new topology was developed, one had to adapt all algorithms and rewrite the software. In contrast, for sequential machines widely used model of the system (von-Neumann-model [BGvN89]) and architecture independent programming languages (e. g., ISO-C99 [ISO05]) exists, so it is easy to develop software that runs efficiently on all computers that follow the model.

Several parallel computer models try to play a similar role for parallel systems. Some are mainly used for theoretical analysis, but others are supported by libraries for the efficient implementation of algorithms on different target architectures.

For the success of parallelism on the market, a general parallel model is needed that is easy to use and allows the development of portable but yet efficient algorithms. Furthermore, a toolchain supporting the implementation is needed, including languages, compilers, libraries, and development tools like parallel debuggers and profilers.

1.2 Contributions of this Thesis

Skillicorn and Talia conclude their survey of parallel algorithms by the following perspective:

> "Thus we can hope that, within a few years, there will be models that are easy to program, providing at least moderate abstraction, that can be used with a wide range of parallel computers, making portability a standard feature of parallel programming, that are easy to understand, and that can be executed with predictably good performance."
>
> *Skillicorn and Talia, 1998[ST98]*

In our view models which have these properties already exist. The main objective of this thesis is to demonstrate that the bulk synchronous parallel (BSP) models are flexible enough to allow portable, yet efficient parallel programs for a wide range of machines: from tightly coupled parallel computers on a single chip to workstation clusters using traditional local area networks.

This thesis covers the entire work flow: parallel computation models, algorithms, and implementations on different parallel architectures. We first explore several of the existing models to find candidates for a general parallel model. We then concentrate on the family of bulk synchronous models, as we think they are good candidates for parallel models for general use like the von-Neumann model for sequential computers. Next, we investigate at algorithms for the BSP model. The models should be in use, i. e., there should be a lot of algorithms in the literature for them. Furthermore, we want practical usable models, so we investigate universal implementations for them.

Our main contributions are the design of a configurable, efficient parallel system on a chip suitable for BSP — demonstrating how the model can guide the development of efficient hardware for general use — and an efficient implementation of the BSP model

for workstation clusters providing only unpredictable usable idle times. Our contributions demonstrate that the BSP model can be used for a wide range of parallel systems. It is thus well suited as a candidate for a commonly used parallel model.

BSP for Parallel System on Chip. We design a hierarchical parallel architecture, with clusters of processors using shared memory and an on-chip interconnection network between the clusters. We develop communication protocols, a BSP library, and a set of simulators to evaluate the system and the influence of different parameters like cache sizes, organizations, et cetera. We evaluate the system using different benchmarks, from simple communication benchmarks to applications like sorting and solving 3-satisfiability, covering a wide range of limiting factors for parallel programs.

BSP for Heterogenous Workstation Cluster. We implement and evaluate a library for efficient execution of BSP programs for heterogenous workstation cluster. As we are allowed to use only the idle times of the computers, and the behavior of other users is not known in advance, we have to adapt to changes of the free capacity online.

For this purpose we implement virtual processors, which can be migrated to idle nodes in the network. To the best of our knowledge, this is the first realization of migrating Linux processes in the userspace, without modifications of the operation system. This is very important, because normally users are not allowed to change the systems of other users. Furthermore, patching the Linux needs updates to the software for every new Linux version. These migration technique is also of independent interest.

We examine the usage of a typical Linux workstation to create models for the load and the idle times of workstation clusters. Based on the load models we implement and evaluate load balancing strategies.

1.3 Related Work

Much research has been done in the field of parallel models and algorithms. We give a survey of some models that are related to the BSP model, and a survey of algorithms and implementations in the following chapters.

In the field of implementations there are different related publications: other BSP implementations (e. g., Oxford BSPlib), other systems for process migration and load balancing, and other approaches for parallel systems on a single chip. We discuss this work in the corresponding chapters.

1.4 Organization

In **Chapter 2** we describe parallel models in the field of bulk synchronous parallel computing. We start with the famous PRAM model and show the development to the bulk synchronous parallel model and its extentions.

Chapter 3 gives a survey over algorithms for the bulk synchronous model to show that it is widely used for algorithm design. The algorithms described in the literature are mostly analyzed using the $\mathcal{O}$ notation (Landau notation), which shows the asymptotic behavior for very large input sizes.

In **Chapter 4** we show implementations, i. e., practical usable software, to confirm that the model is not only useful for analyzing algorithms in the $\mathcal{O}$ notation but also leads to fast algorithms in practice. Furthermore, we give a survey of libraries for implementation of BSP algorithms. There are many more algorithms than implementations, mainly because the model is widely used in theory but still unknown to many programmers, which will hopefully change in the future. The last section in that chapter introduces our system to build libraries of algorithms, and shows a way to automatically configure optimal algorithms for a given machine and a given problem size using a collection of annotated algorithm implementations.

The next two chapters deal with our implementation of libraries for BSP computing for two very different architectures: **Chapter 5** describes a parallel system on a single chip, with low latencies, high bandwidth, but small local on-chip memories. It shows the flexibility of the model, as the same algorithms, in our implementations even nearly with the same source can be used for quite different systems. It also demonstrate how BSP can support the development of hardware architectures, e. g. designing communication protocols suitable for BSP.

In **Chapter 6** we propose an efficient implementation for a parallel architecture at the other end of the range of parallel systems: a lossly coupled workstation cluster with some external load, where we can only use unpredictable idle times. We show how the migration of virtual processors can help to cope with load changes and faulty machines.

Publications

Parts of this work have been published on conferences and in journals. A fast BSP algorithm for factorization of polynoms is evaluated in [BvzGG+01]. The combination of algorithms and implementation and its automatical configuration is introduced in [BMW02]. Some issues of the design and implementation of our BSP implementation (PUB library) are described in [BJvR03], the migration and load balancing functionality for heterogenous workstation clusters in [Bon07]. A model for the utilization of workstations (constant-usage-interval load model) is published in [BGM05, BGM06]. An introduction to the System-On-Chip architecture is [BSR+03].

Chapter 2

Parallel Models

In this chapter we will give a survey of parallel computing models. We will start by introducing a family of parallel random access machine models [SS63], which are natural extensions to the sequential random access machine. Next bulk synchronous models will be described, and we conclude by reviewing some other related models. We concentrate on these models, because we think they can act as general parallel models, which are suitable for theoretical analysis as well as for portable and efficient implementations.

Figure 2.1 gives an overview of the parallel models described in the following paragraphs and their relations. In [LMR95], Li, Mills, and Reif present a framework of using resource metrics to characterize models of parallel computation. They concentrate on the resources that are accounted for by the model, for example, memory organization, latency, or block transfers. The paper describes a number of parallel models and classifies them with respect to their resources.

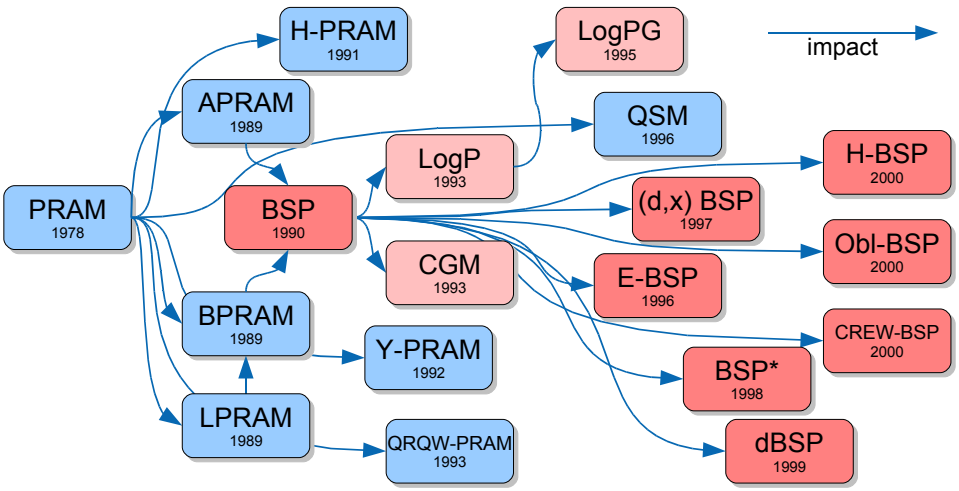
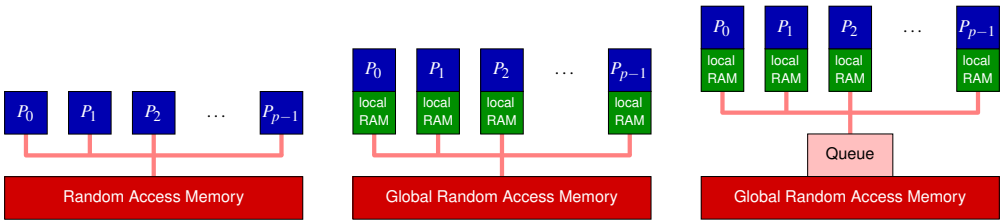


Figure 2.1: A Map of some Models for Parallel Computing



(a) PRAM (b) LPRAM, BPRAM (c) QRQW-PRAM

Figure 2.2: Parallel Random Access Machine

Another survey including parallel languages and systems with implicit parallelism as well, has been written by Skillicorn and Talia [ST98]. The authors use a set of six criteria that in their view an ideal model should satisfy. Four criteria are related to the needs of software developers, two address the needs for actual execution of the models on real parallel machines. The main focus of the paper is the classification of the models in categories based on these criteria.

2.1 Parallel Random Access Machine models

In this section we will describe a family of parallel computing models that use a global shared memory. We will start by introducing the standard Parallel Random Access Machine (PRAM) model, then continue by adding a number of extensions including additional local memory and block transfers.

2.1.1 Parallel random access machine (PRAM) model

The *parallel random access machine* [FW78] is an extension of the sequential random-access-machine (RAM). A RAM consists of one processing element (CPU) and a random access memory. The CPU can read and write to arbitrary cells of the memory.

In a PRAM, there are p processors that can read and write arbitrary cells of a global shared memory. The processors have a central clock and execute each instruction of a given program synchronously. All communication between the processors is done by writing to and reading from the memory (Figure 2.2). Many variants of PRAMs exist in the literature. They differ in how simultaneous accesses of different processors to the same memory cell are handled: *Exclusive* read and write (EREW) does not allow any conflicts. *Concurrent* read/write (CRCW), as the most powerful version, can handle such accesses. For CRCW PRAMs, there are different rules for resolving write conflicts, including *common* (all processors have to write the same), *priority* (processor with lowest id writes), and *arbitrary* (the winning processor is not defined).

Communication costs are not accounted for by the model. Thus, it is suited to analyze the maximal degree of parallelism which is possible for a given problem while neglecting problems occurring in practical applications, for instance, limited bandwidth or congestion.

2.1.2 Local-Memory PRAM (LPRAM)

The *Local Memory PRAM* introduced by Alok Aggarwal, Ashok K. Chandra and Marc Snir [ACS90] includes local memory. In addition to the global memory, each processor has its own unlimited private local memory. At each time step, a processor can either read or write a word to the global memory, or perform a local computation step. The access to the global memory is modelled by concurrent read, exclusive write.

The authors provide proves for upper and lower bounds for the LPRAM model for three problems: matrix multiplication, sorting, and computing an n-point FFT graph.

2.1.3 Block-PRAM (BPRAM)

The *Block-PRAM* model [ACS89] is an extension of the LPRAM, it includes block by block transfers. All accesses to the shared global memory are blockwise. This means that a processor can transfer a block of consecutive memory cells from the global memory to the local memory. The same is possible in the opposite direction. Copying of data is charged by costs of $b+l$, where b is the size of the block and l is the latency (a parameter of the machine). In the BPRAM model, different accesses are not allowed to overlap, i. e., the shared memory is modelled by exclusive read, exclusive write (EREW). Minimal running time and the total work performed by the processors are important measures of the algorithm performance in this model.

The authors demonstrate the capabilities of their model by analyzing sorting, global sum, matrix transposition and multiplication, rational permutations, and Fast Fourier Transformation. Furthermore, they summarize the relationship of execution time of standard PRAM and BPRAM with the following equation, since one step of a EREW-PRAM can be simulated with l steps of the Block-PRAM:

$$\mathrm{T}_{\text{EREW-PRAM}} \leq \mathrm{T}_{\text{BPRAM}} \leq l \cdot \mathrm{T}_{\text{EREW-PRAM}}$$

2.1.4 Queue-Read Queue-Write PRAM

The *Queue-Read Queue-Write (QRQW) PRAM* was introduced by Gibbons, Matias, and Ramachandran [GMR98] and is a model situated between the EREW- and the CRCW PRAM. Similar to the LPRAM, each processor has its own local memory in addition to the global memory. Concurrent access to the global memory is allowed, but the cost

of a step is increased by the number of processors that access the global memory. The motivation of this cost function is that all requests are stored in a queue and processed sequentially. In [GMR98] the relationship to other models is also discussed: a p processor QRQW-PRAM can be simulated on a $p/\log p$ processor BSP computer (cf. Section 2.2) with slowdown $\mathcal{O}(\log p)$ with high probability.

2.1.5 Asynchronous PRAM (APRAM)

The *Asynchronous PRAM* by Phillip B. Gibbons [Gib89] is a PRAM that also incorporates both local and global memory. However, in contrast to the other PRAM models, each processor of the APRAM has its own local clock and executes instructions of its own program independently of the timing of other processors. In addition to the global and local read and write operations, there is an instruction to synchronize arbitrary subgroups of the processors. All processors in the subgroup stop until the last has reached a synchronization barrier.

To prevent race conditions, i. e., the output is depended on the order of events, processors are prevented from reading a global memory cell that has been modified by another processor until a synchronization event has occurred. The cost model assumes a global clock, i. e., all instructions on all processors are equally expensive. However, an algorithm for the APRAM model is considered correct only if it works regardless of any delay that may occur.

2.2 Bulk Synchronous Parallel Models

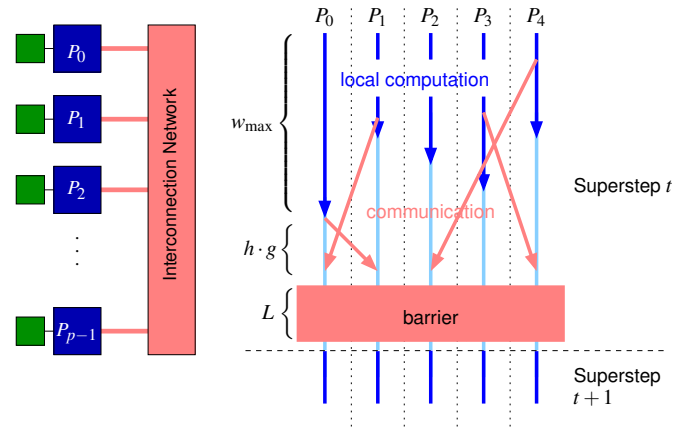
In this section we will describe the Bulk Synchronous Parallel Model and its extensions, as we think they are the right candidates for a general parallel model, which is usable for theoretical analysis as well as for portable and efficient implementations.

The models are more restrictive concerning communication patterns than the PRAM models in the previous section, but this simplifies the analysis of algorithms as well as the implementation of the model in real hardware systems.

2.2.1 The Bulk Synchronous Parallel Model

The *Bulk Synchronous Parallel Model* (BSP) was introduced by Leslie G. Valiant as a bridge between the hardware and the software to simplify the development of parallel algorithms.

> "A major purpose of such a model is simply to act as a standard on which people can agree."
>
> *Leslie Valiant, 1990 [Val90]*



(a) BSP Computer (b) Superstep

Figure 2.3: BSP Model

On the one hand it provides an abstract view of the technical structure and the communication features of the hardware to use (e. g., a parallel computer, a cluster of workstations, or a set of personal computers (PCs) interconnected by the internet) for the developer. On the other hand it helps hardware designer to build efficient parallel machines without knowing the applications running on it in advance. There are three different parts of the model: a *machine model*, a *programming model* and a *cost model*. There are several views of this model with slightly different descriptions, but they are all equivalent up to constant factors in the cost model. We start with our view, based on our experiences with implementations. Afterwards, we describe the differences.

Machine model. A *BSP computer* is defined as a set of processors with local memory and an arbitrary communication mechanism (e. g., a network or shared memory) that supports point-to-point communication between the processors, and barrier synchronization. The model does not assume a specific topology or exploit locality (cf. Figure 2.3(a)).

Programming Model. A *BSP program* consists of a set of *BSP processes* and a sequence of *supersteps* – time intervals bounded by the barrier synchronization. Within a superstep each process performs local computations and sends messages to other processes; afterwards it indicates by calling the synchronization method that it is ready for the barrier synchronization. When all processes have invoked the synchronization method and all messages are delivered, the next superstep begins. Then the messages sent during the previous superstep can be accessed by its recipients.

Cost Model. The cost of a superstep is the sum of $w_{\max}$, the maximal local work of a processor, h, the maximal amount of data that is sent or received by one processor multiplied with the *gap* g of the network, and the time L (*latency*) for one barrier synchronization (Figure 2.3(b)):

$$\text{Cost of a superstep} = w_{\max} + hg + L$$

A communication pattern where each processor sends or receives at most h bytes is called *h-relation*.

In the original description of Valiant [Val90] the synchronization hardware checks every L time steps whether all processors have finished the superstep. Thus, the length of a superstep is always a multiple of L. The difference in the cost function compared to our definition is at most L times the number of supersteps, thus at most a factor of 2.

Computational power. The computational power of BSP is analyzed by Martin Beran in [Ber98]. He assumes an unlimited number of processors, and functions $g(p)$ and $L(p)$ describing the bandwidth and the latency with respect to the number of active processors. If $g(p)$ and $L(p)$ are polylogarithmic in p, a BSP computer is asymptotically as powerful as a PRAM. They both belong to a class C_2, which are all machine models that are polynomial-time equivalent to the space complexity of Deterministic Turing Machines (DTM). All machines in C_2 can solve all problems of the class $\mathcal{PSPACE}$ in polynomial time (with unbounded number of processors).

For $g(p) = \Omega(p^a)$, or $L(p) = \Omega(p^a)$, $a > 0$, or p constant, a BSP-Computer belongs to the Class C_1 (all machines that are polynomial-time equivalent and linearly space equivalent to DTM).

More details on the complexity classes C_1 and C_2 can be found in the Handbook of Theoretical Computer Science [vL90].

Extensions to BSP. A lot of extensions to the standard BSP model have been proposed since 1990, for example, E-BSP, BSP*, dBSP, Obl-BSP, or H-BSP. Some of them try to improve the accuracy of running time predictions by using more parameters to describe the machine, for example by introducing a minimal block size for efficient communication in the BSP* model. Others improve the power of the machine by introducing new features, an example is oblivious synchronization.

2.2.2 BSP*

Most real machines need some time to prepare messages, for example, for creating message headers or starting the transmission. So the achieved bandwidth for the communication depends on the size of the messages. In the BSP model the cost depends only on the sum of all transmitted data (h relation), not on the size of the messages. The BSP* model

by Armin Bäumker, Wolfgang Dittrich, and Friedhelm Meyer auf der Heide [BDM98] introduces an additional machine parameter B. Informally, B is the minimal size a message must have to achieve a "good" network bandwidth. More formally, the communication cost of a superstep is $h \cdot g$, where h is the maximal amount of data h_i, $i = 0, \ldots, p-1$ send or received by a processor. In opposite to the normal BSP model, these h_i are the sum of all message sizes where each message smaller than B bytes is charged for B.

The authors use multisearch as an example to show the improvement over BSP.

2.2.3 Oblivious BSP

One often noted objection to the use of BSP is the global synchronization. Especially for large parallel machines having many processors, the cost of a global synchronization seems to be quite big (although in most implementations it is of order $\mathcal{O}(\log p)$ for p processors). So there are some proposals to reduce this cost. The Oblivious-BSP model by Jesus A. Gonzalez, Coromoto Leon, Fabiana Piccoli, Marcela Printista, José Luis Roda, Casiano Rodríguez, and Francisco de Sande [GLP+00] formalized *Oblivious Synchronization*, which was first implemented in the BSPk system (s. Section 4.1.1).

In the BSP model all processors are in the same superstep, guaranteed by the barrier synchronizations between supersteps. But for a correct execution it is only needed that messages are received at the right time, i. e., two processors that do not communicate with each other do not have to synchronize. In the Oblivious BSP model processors may be in different supersteps, the only limitation is that a processor is not allowed to enter a new superstep until all messages for it have been received. As it is not possible for the system to know in advance if other processors, which are still in earlier supersteps, will send messages, the user has to specify the number of messages each processor will receive.

The processor waits until the messages have been received and continues with the next superstep. This can reduce the idle time waiting for other processors in barrier synchronizations, especially if the amount of work in a superstep is not equal on all processors. Furthermore, the time for the barrier synchronization is saved.

The authors demonstrate the accuracy of running time predictions in their model by evaluating a nested parallel Discrete Fast Fourier Transform algorithm with the PUB library (Section 4.1.2) on a Cray T3E supercomputer. The performance gain of the oblivious synchronizations in practice can be seen in [BJvR03], where the running time of a multi search program is evaluated using different advanced features of the PUB library.

2.2.4 E-BSP

In 1996 Ben H. H. Juurlink and Harry A. G. Wijshoff proposed the *Extended BSP (E-BSP) model* [JW96]. It includes unbalanced communication and locality. The authors extend the cost model by replacing the term h-relation in the BSP cost model by an

(M,k_1,k_2)-relation, where M is the total number of messages, k_1, k_2 is the maximal number of message a single processor sends respectively receives, thus a (full) k-relation is a (kp,k,k)-relation in the E-BSP model.

The locality of a superstep is defined as the maximal difference of the identifiers of any sender/receiver pair. The implementation should use a numbering of the processors, such that two processors with a small difference in their identifiers have a short distance in the underlying physical network, for instance, using a peano embedding for a mesh of dimension two. The PUB library uses such embeddings for some mesh architectures, for instance, the Parix GCel/GCPP.

A comparison with other models can be found in Ben Juurlink's PhD thesis [Juu97], including comparisons of communication times of sorting algorithms for the E-BSP and for the standard BSP model.

2.2.5 Decomposable BSP (dBSP)

The *Decomposable BSP model* by Martin Beran [Ber99] is another approach to include locality. A dBSP computer can dynamically partition itself into smaller parts which behave as dBSP computers with smaller p. Communication between processors of different partitions is impossible. The parameters g and L of the original BSP model are replaced by functions of p. In most implementations, $g(p)$ and $L(p)$ increase with p. If the synchronization is implemented using message passing and a tree as communication structure, $L(p)$ is usually of order $\mathcal{O}(\log p)$.

The paper shows a slowdown of $\mathcal{O}(\max\{g(p),L(p)\})$ for a superstep of a dBSP computer simulated on a BSP computer, and an additional cost of $\mathcal{O}((1+L(p)+g(p))\cdot\log p)$ for the partition and join operations.

Besides the performance advantage, the model is useful in practice, if different algorithms should run simultaneously on parts of a parallel computer. The PUB library supports this model: it is possible to partition the machine into independent BSP computers; furthermore, BSP parameters of the machine are given for the actual size of the partition.

2.2.6 H-BSP

The *Heterogeneous Bulk Synchronous Parallel (HBSP) model* by Tiffani L. Williams and Rebecca J. Parsons [WP00] is a generalization of the BSP model. The gap g is substituted by the gap g_i, $i = 0,\ldots,p-1$ of each processor. The parameter g_i describes the rate at with a processor can send data to the network. Additional parameters c_i, $i = 0,\ldots,p-1$, define the computation power of the processors, relative to the slowest one.

The cost function is adapted, such that the local work is weighted by the c_is. The term $g\cdot h = g\cdot\max_{0\le i<p}\{h_i\}$ for the communication cost is replaced by $\max_{0\le i<p}\{g_i h_i\}$,

where h_i is the amount of data sent to the network by processor i.

How to use the model for designing algorithms is shown by some examples: prefix sums, matrix multiplication, and randomized sample sort.

2.2.7 (d,x)-BSP

The *(d,x)-BSP model* by Guy E. Blelloch et al. [BGMZ97] is an extension of the BSP model designed for high performance parallel computer with global shared memory. The additional parameters d (delay) and x (expansion) describe the timing of the memory, i. e., the time for an access (typically in the order of 10 clock cycles for machines in 1996) and the ratio of the number of processors and the number of parallel accessible memory banks. The cost functions is expanded by a term for the maximal number of accesses to a memory bank, scaled with the delay d.

Typical example machines for this model were NEC SX3 ($p = 4$, $x = 256$), Cray C90 ($p = 16$, $x = 64, d = 6$), or Cray J90 ($p = 16$, $x = 64, d = 14$).

2.2.8 CREW-BSP

The standard BSP model allows point-to-point messages and barrier synchronizations only. Sometimes the hardware supports other communication patterns, for example, broadcasts are easy to realize if the communication medium is shared, for example a bus.

The *CREW-BSP-model* (concurrent read exclusive write) by Michael T. Goodrich [Goo00] extends the standard BSP-model by arbitrary multicast operations, i. e., the destination of a message is a set of processors. The same paper defines also a CRCW-BSP computer, which is able to combine data from more than one processor using an arbitrary relation. This computer is more powerful than a standard BSP-computer, for instance, it can compute the maximum of p numbers in constant time.

2.3 Other Parallel Models

In this section we describe two parallel models, which are similar to the BSP model, but does not divide the time into supersteps, thus, they are not synchronous models.

2.3.1 Coarse grained multicomputer (CGM)

The *Coarse Grained Multicomputer (CGM)* was introduced 1993 by Dehne, Fabri, and Rau-Chaplin [DFRC93, DFRC96]. It is the first model of a family of parallel computation models called coarse grained parallel computing (CGP) and is similar to the LogP model (which we will describe in the next section) and the BSP model. A CGM(n, p)

consists of p processors with $\mathcal{O}(\frac{n}{p})$ local memory each, and an arbitrary interconnection network. Is is said to be "coarse grained" because the size of the memory is defined to be "considerably larger than $\mathcal{O}(1)$". Dehne et al. assume that $\frac{n}{p} \geq p$.

The basic idea behind this model is the following: The algorithms partition the given problem into p subproblems of size $\mathcal{O}(\frac{n}{p})$. These subproblems are solved sequentially by the processors; afterwards the results are combined using a global communication operation. This scheme is iterated; for many problems a constant number of iterations is sufficient. For the global operation sorting is used. Many parallel architectures have global sorting as a system call, or a highly optimized implementation can be obtained as public domain software. Other communication operations, for example, broadcast (segmented, multinode), total exchange, or partial sum (scan) can be implemented using a constant number of global sort operations. So in the CGM sorting is the only way to communicate, and the cost for an algorithm is based on $T_S(n,p)$, the time needed to sort n data items on p processors.

2.3.2 LogP

The *LogP model* was introduced by Culler, Karp, Patterson, Sahay, Schauser, Santos, Subramonian, and von Eicken [CKP+93] in 1993 and tries to model a processor network by four parameters: L is the *latency*, i. e, the delay for sending a very small message from a source to a destination, o is the *overhead*, the time a processor is busy starting a transmission or reception of a message, including the time needed for computing message headers. Parameter g is the *gap*, the minimal time interval between consecutive message transmissions or consecutive message receptions, the reciprocal is the per processor bandwidth. P is the number of processors. Thus, LogP does not consider the topology of the network, similar to the CGP model. Furthermore, the capacity of the network is limited, i. e., at most $\lceil L/g \rceil$ data can be in the network at the same time.

On the one hand this model is more general than CGM or BSP as it allows arbitrary asynchronous communication and does not restrict the algorithms to global communications or supersteps, on the other hand proofs of correctness or running time analysis of algorithms may be much complexer.

2.4 Conclusion

We described parallel models, from the PRAM to BSP and its extensions. We think that the BSP model and its extensions are a good way to model parallel systems such that algorithm engineers can develop parallel algorithms that run fast on a wide area of parallel systems, because: On the one hand the PRAM model is hard to realize, because multiple accesses to a global shared memory require an expensive cross bar system. On the other hand the plain BSP model is too restricted, i. e., there are algorithms that does

not fit into BSP very well, but some relaxations, for instance, oblivious synchronization or a decomposable machine increase the number of possible applications.

In some areas it may even be needed to soften the model a little, to allow asynchronous messages in some parts, but at the cost of loosing some nice features of the synchronous model, for example, simplicity of verification. Our implementation, the PUB library, which is described in Section 4.1.2, follows that direction.

Chapter 3

Algorithms

A suitable model for parallel computing should support the development of efficient algorithms. This chapter starts with an overview of algorithms for the bulk synchronous setting. In the second part we will propose a system which can be used to build libraries of algorithms for more complex problems by combining basic algorithms. It configures an optimal algorithm based on information of the input (e. g., the size of the input), and the architecture, i. e., the BSP parameters, using a library of BSP algorithms and their running time descriptions.

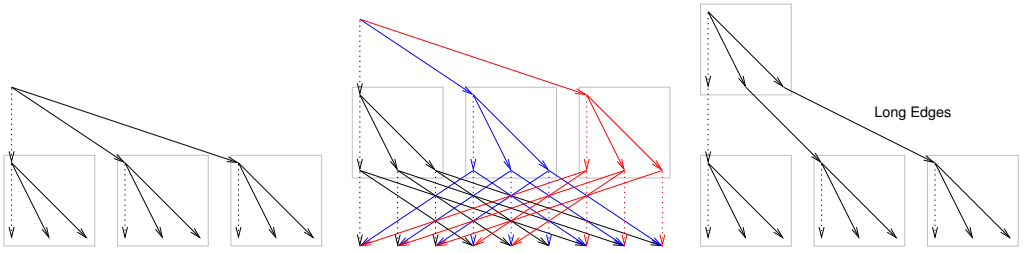
3.1 BSP-Algorithms

In this section we will give a short overview of algorithms for the BSP model to show that it is widely used in algorithm design. Here were focus on algorithms and their theoretical analysis in the BSP model. In Section 4.2 we will show some implementations of BSP algorithms.

There are many BSP algorithms for various different application areas. Simple examples are basic communication functions (e. g., broadcast or parallel prefix operations, which are already mentioned in the paper introducing the BSP model [Val90]) and sorting. Examples for more complex applications are graph algorithms, optimizations, simulations, and algebra.

3.1.1 Basic Communication Algorithms

The BSP model supports point-to-point messages and synchronizations only. All other communication pattern, for instance *broadcast* (one processor sends a message to all others), *multicast* (one message to a set of destination), *total exchange* (every processor has data for all other processors), or the parallel prefix operations *scan* and *reduce* have to be realized on top of these point-to-point messages.



(a) TREE$_{d=3}$ (b) SCATTERGATHER$_{k=3}$ (c) ROOT

Figure 3.1: Broadcast Algorithms

Some implementations of BSP are more powerful and support such advanced communication functions directly, for example, the PUB library (Section 4.1.2), which is sometimes more efficient [Rie00], and more comfortable for the user. There are also models that allow this, like CREW-BSP (Section 2.2.8), but they are not as widely used as other BSP models.

Broadcast

In the *Broadcast* problem one processor has one message of size m that has to be sent to all other processors. A basic algorithm TREE is given by the author of the BSP model [Val90]: in $\log_d p$ supersteps each processor that knows the message sends it to d others. As d messages of size m are sent in each superstep, a superstep takes time $\mathcal{O}(dm \cdot g + L)$, thus the total time is $\mathcal{O}((dm \cdot g + L) \log_d p)$.

Other broadcast algorithms can benefit from locality, for example in the dBSP model. An example for such an algorithm is ROOT [JKMR03]: the message is sent to $\sqrt{p}$ subgroups in the first step, and distributed inside these groups by a recursive broadcast call.

In [Rie00] other broadcast algorithms are analyzed: PTREE (pipelined version of the standard algorithm TREE), and SCATTERGATHER, an algorithm that splits the message in k parts, broadcasts the parts to subgroups, and then sends the parts from a subgroup to all other nodes (similar to an all-to-all communication). These algorithms are illustrated in Figure 3.1, their running time in the dBSP model can be found in Table 3.1.

3.1.2 Parallel Prefix Operations

In the *Parallel Prefix* problem every processor i has some data x_i, and for a given associative operation $\otimes$ one needs to compute $x_0 \otimes x_1 \otimes \cdots \otimes x_i$ for all $i = 0, \ldots, p-1$. This problem is sometimes referred to as *scan*. A simpler problem is called *reduce*: instead of computing the results for all i, only $\bigotimes_{i=0}^{p-1} x_i$ is needed. A simple tree based algorithm with

Table 3.1: Running Times of Broadcast Algorithms in the dBSP Model [Rie00]

Algorithm	Result
Tree	$\sum_{i=0}^{\lceil \log_d p \rceil - 1} \left((d-1) \cdot (m \cdot g(\left\lceil \frac{p}{d^i} \right\rceil) + L(\left\lceil \frac{p}{d^i} \right\rceil) \right)$
PTree	$k \cdot \left(L(p) + 2\lceil \frac{n}{k} \rceil \cdot g(p) \right) + \sum_{i=1}^{\lceil \log p \rceil - 1} L\left(\frac{p}{2^i}\right) + 2\left\lceil \frac{m}{k} \right\rceil \cdot g\left(\frac{p}{2^i}\right)$
ScatterGather	$2(m \cdot g(p) + L(p)) +$ time for broadcast of $\frac{m}{k}$ to $\frac{p}{k}$ processors
Root	$\sum_{i=0}^{\lceil \log \log p \rceil} 2^i \left(m \cdot g(\left\lceil p^{1/2^i} \right\rceil) + L(\left\lceil p^{1/2^i} \right\rceil) \right)$

Table 3.2: Running Times of Sorting Algorithms

Algorithm	Result	Reference
CommunEfficientParSort	$\mathcal{O}\left(\frac{n \log n}{p} + \frac{\log n}{\log(n/p)} \left(\frac{n}{p} \cdot g + L \right) \right)$	[Goo00]
Randomized SampleSort	$\mathcal{O}\left(\frac{n \log n}{p} + (p^\varepsilon + \frac{n}{p}) \cdot g + L \right)$ w. h. p.	[GV94]
Regular Sampling	$\mathcal{O}\left(\frac{n \log n}{p} + \frac{n}{p} \cdot g + L \right)$ for $n \geq p^3$	[SS92]
XY-Sort	$n = \sqrt{p}$, $\mathcal{O}(\sqrt{p} \cdot g + L) + T_{\text{red}}$ T_{red} time for reduce on $\sqrt{p}$ proc.	[Rie00]

a tree of degree d archives the same order of running time than the broadcast algorithm TREE.

3.1.3 Sorting Algorithms

Sorting is one of the basic problems in computer science. It is well understood and there are sorting algorithms for almost all parallel computing models. Some results for BSP sorting algorithms are summarized in Table 3.2.

Sample Sort. Parallel SAMPLESORT is an extention of the sequential QUICKSORT algorithm. QUICKSORT splits the input into two parts, elements smaller than a splitting element and those that are greater. Next these two groups are sorted recursively. In SAMPLESORT, the data is partitioned into p groups according to $p-1$ *splitters*. Each processor sends the keys to the right destination, next the received keys are sorted locally. The main problem is to find good splitters such that the sizes of the partitions are nearly equal. For this purpose samples of the input are selected, sorted, and used as splitting elements. The main part of the analysis is about the number of samples. An algorithm based

Table 3.3: All-pairs Shortest Paths Algorithms [Tis01]

Algorithm	**Result**
Dijkstra / Path Doubling	$\mathcal{O}(\frac{n^3}{p} + \frac{n^2}{p^{2/3}} + (\log p)L)$,
	$\mathcal{O}(\frac{n^3}{p} + n^2 + L)$
Floyd-Warshall-algorithm	$\mathcal{O}(\frac{n^3}{p} + \frac{n^2}{p^\alpha}g + p^\alpha L)$, $\frac{1}{2} \leq \alpha \leq \frac{2}{3}$

on this approach, proposed by Gerbessiotis and Valiant [GV94], needs a running time of $\mathcal{O}\left(\frac{n\log n}{p} + (p^\varepsilon + \frac{n}{p}) \cdot g + L\right)$ for sorting n keys, with high probability for any constant $0 < \varepsilon < 1$, if $p \leq n^{1-\delta(\varepsilon)}$ where $\delta(\varepsilon)$ is a small constant depending only upon ε.

The randomized sampling can be replaced by regular sampling. This technique is described by Hanmao Shi and Jonathan Schaeffer [SS92].

XY-Sort. XY-SORT is a sorting algorithm used as an example in [Rie00]. It sorts $\sqrt{p}$ keys on p processors with a constant number of communication operations: the keys are distributed using two multicast operation, compared in constant time, and the total rank of each key is calculated with $\sqrt{p}$ parallel reduce operations, each on a subgroup of size $\sqrt{p}$.

3.1.4 Graphalgorithms

There are many algorithms for problems based on graphs in the literature. The challenge for the algorithm designer is that the input is not as regular as in other application areas like matrix multiplication, thus the algorithms are almost not oblivious, i. e., the communication pattern is dependent on the input.

Shortest Path. There are two main sequential algorithms for shortest path problems: Dijkstra's algorithm and the algorithm of Floyd-Warshall. For the *All-Pairs-shortest Paths problem* one can execute DIJKSTRA algorithm in parallel for all sources, or use a parallel variant of FLOYDWARSHALL, which is based on matrix potentialization. This algorithm is more communication efficient, but needs more than a constant number of supersteps. Table 3.3 summarizes the results of Alexandre Tiskin [Tis01].

Minimal Spanning Tree. We give a formal definition of the problem in Section 3.2.2. There are various parallel algorithms for the calculation of a minimal spanning tree. A communication lower bound and an overview of BSP algorithms can be found in the PhD thesis of Ingo Rieping [Rie00]. The algorithms are made up of some modules: BSP-MERGE, which merges two minimal spanning trees,and BSPBORUVKA, which executes

Table 3.4: Running Times of MST Algorithms

Algorithm	Result	Reference
MSTMerge	$\mathcal{O}\left(T_{\text{seq}}(n,\frac{m}{p})+\frac{m}{p^{1+\varepsilon/2}}+L\right)$ for $p^{1+\varepsilon}\leq m/n$	[Rie00]
MSTDense	$\mathcal{O}\left(n\log\log p+\frac{m}{p}+n\cdot g+\log p\cdot L\right)$ for $p\leq\frac{m}{n}$, 1-optimal if $p\log\log p=o(\frac{m}{n})$	[ADJ+98]
MSTSparse	$\mathcal{O}\left(\frac{n\log p+m}{p}+\frac{m}{p}\cdot g+\frac{(\log p)^2}{\log((m+n)/p)}L\right)$ with prob. $\geq 1-n^{-c}$ for any $c>0$, $p\leq m+n$, $\log p\leq\frac{m}{n}\leq p$, $m\geq n\log n$	[ADJ+98]
BSP-AS	$\mathcal{O}\left(\frac{m+n}{p}\log p+\frac{m+n}{p}\log p\cdot g+\frac{(\log p)^2}{\log((m+n)/p)}L\right)$ for $n+m\geq p$	[ADJ+98]

so-called Borůvka steps. In each step a (super-)vertex selects a cheapest outgoing edge and joins the two (super-)vertices to a new supervertex. A third module is BSP-AS, a BSP version of the algorithm due to Awerbuch and Shiloach, which was originally developed for a CRCW-PRAM. By combining these modules one get efficient algorithms for sparse as well as for dense graphs.

The minimal spanning tree problem is also used as an example for combining several BSP algorithms together to an optimal algorithm, adaptive to the BSP computer and to the input size. The modules BSPMERGE and BSPBORUVKA and their combinations are described in more details in Section 3.2.2.

3.1.5 Optimization

One kind of optimization is searching for an optimal solution in a given space of feasible solutions. Starting from a solution the algorithms perform small legal changes to the instance and try to improve it concerning a given cost function. As the changes are usually chosen by an algorithm with some random choices, a natural parallelization is to start the search in parallel on different processors. After some time the processors exchange their locally best solutions and start searching at the globally best solution found so far. This scheme fits perfect to the BSP model: in each superstep the local search is done, at the end of the superstep the best solution is computed by a parallel prefix operation using max (or min) as combine function. This approach is used by Klaus Brockmann and Ingo Rieping for a parallel tabu search algorithm (unpublished), based on parallel tabu search implementations from Stefan Bock and Otto Rosenberg [BR00].

3.1.6 Simulations

Discrete Event Simulation. In a *discrete event simulation* there are events at certain points of time. These events have to be processed in non decreasing time order. Each

event can change the state of the simulation, and introduce new events.

One approach for a parallel algorithm is to distribute parts of the simulation to the processors and send events concerning parts simulated on other processors by messages. This is implemented by an event queue locally on each processor. For high efficiency these queues cannot be synchronized with each other, i. e., the events processed in parallel may not be in the right order concerning global time. One solution is called *Optimistic Discrete Event Simulation*. Each processor has its own local virtual time (*lvt*) and processes events in its local queue in non decreasing time order. If it receives an event with a timestamp earlier than the own *lvt*, it rollbacks the simulation (time warp) and restart it at that time, sending *anti-messages* for each wrong message it has sent in the past.

Several bulk synchronous parallel algorithms for optimistic discrete event simulation are proposed by Radu Calinescu [Cal96].

N-Body. The *N-body problem* is the problem of simulating the movement of a set of bodies (or particles) under the influence of some type of force, for example gravitational or electrostatic. A basic approach is to simulate the system by advancing the bodies in discrete time steps. In each time step, the algorithm computes (or approximates) the force exerted on each body due to all other bodies; this determines the acceleration and speed of that body during the next time step.

Efficient parallel implementation of $\mathcal{O}(N)$ adaptive tree codes for the BSP model are given by David Blackston and Torsten Suel [BS97].

3.1.7 Algebraic Problems

Algebraic problems have been addressed by parallel algorithm designers since long ago. Operations on matrices, for instance, multiplication of two matrices, are often used as an introduction to parallel algorithms. Many algorithms are oblivious, they have a static communication behavior that simplifies things like load balancing. The results for the following algorithms are summarized in Table 3.5.

Matrix Multiplication. There are different algorithms for *matrix multiplication*. A BSP version of the standard algorithm is given as a simple example of a tightly synchronized algorithm by Valiant [Val90], a faster algorithm using the idea of Strassen to reduce the work is given by William F. McColl [McC95a].

McColl and Tiskin propose other algorithms optimizing the memory use. In [MT99] they analyse memory efficient BSP algorithms for both the standard and the fast matrix multiplication.

Solving Linear Equations. A system of linear equations over a domain $\mathcal{D}$ is given by a matrix $A \in \mathcal{D}^{n \times m}$ and a vector $b \in \mathcal{D}^n$. The problem is to find one or all solutions $x \in \mathcal{D}^n$

Table 3.5: Algebraic BSP algorithms

Problem	**Algorithm**	**Result**	**Reference**
Matrix Multiplication	naive	$\mathcal{O}(\frac{n^3}{p} + \frac{n^2}{\sqrt{p}} \cdot g + L)$	[Val90]
	standard	$\mathcal{O}\left(\frac{n^3}{p} + \frac{n^2}{p^{2/3}} g + L\right)$, $p \leq n^2$	[McC95b]
	Strassen	$\mathcal{O}\left(\frac{n^{\log_2 7}}{p} + \frac{n^2}{p^{2/\log_2 7}} \cdot g + L\right)$, $p \leq n^{2\log_2 7/(2+\log_2 7)}$	[McC95a]
Dense Matr.$\times$Vec.	standard	$\mathcal{O}\left(\frac{n^2}{p} + \frac{n}{\sqrt{p}} g + L\right)$, $p \leq n$	[McC95b]
Sparse Matr.$\times$Vec.[1]		$\mathcal{O}\left(\frac{cn}{p} + n + \left(1 - \frac{1}{p}\right) n \cdot g + L\right)$	[Bis04]
Gaussian Elimination	block rec.	$\mathcal{O}\left(\frac{n^3}{p} + \frac{n^2}{p^\alpha} g + p^\alpha L\right)$, $\frac{1}{2} \leq \alpha \leq \frac{2}{3}$	[Tis07]
with pairwise elem.		$\mathcal{O}\left(\frac{n^3}{p} + \frac{n^2}{p^\alpha} g + p^\alpha L\right)$, $\frac{1}{2} \leq \alpha \leq \frac{2}{3}$	[Tis07]
LU Decomposition		$\mathcal{O}\left(\frac{n^3}{p} + \frac{n^2}{\sqrt{p}} \cdot g + n \cdot L\right)$	[Bis04]
		$\mathcal{O}\left(\frac{n^3}{p} + \frac{n^2}{\sqrt{p}} \cdot g + \sqrt{p} \cdot L\right)$	[McC95b]
		$\mathcal{O}\left(\frac{n^3}{p} + \frac{n^2}{p^{5/8}} \cdot g + n \frac{\log p}{\log(L/g)} \cdot L\right)$, $p \leq n^{8/5}$	[Juu97]
Solution Triangular Lin. System	back substitution	$\mathcal{O}\left(\frac{n^2}{p} + n \cdot g + p \cdot L\right)$	[McC95b]
Fourier Transform.	FFT	$\mathcal{O}\left(\frac{n \log n}{p} + \frac{n}{p} \cdot g + L\right)$, $n \geq p^2$	[Bis04]
Exponentiation in Finite Fields	addition chains	$\mathcal{O}(\log p \cdot (g + L))$	[Nöc01]
Poly. Factorization			[BvzGG$^+$01]

[1] c denotes the number of non-zero elements of the matrix

that satisfy the equation $A \times x = b$. There are different algorithms for solving this equations with different objectives: A parallel version of the standard gaussian elimination is analyzed by Alexandre Tiskin [Tis03, Tis07]. If there is more than one such system with the same matrix A, one could split A into a product of two triangle matrices $A = L \times U$. Knowing L and U, the solution can be computed efficiently for various instances of b. This so called LU-decomposition is one of the examples in [Bis04].

Other Computer Algebraic Problems. Michael Nöcker analyses the problem of exponentiation in finite fields in his PhD thesis [Nöc01] and propose data structures for fast algorithms. As a computation model for the parallel algorithms he uses amongst others

the BSP model. Together with others he also uses BSP for fast factorization of large polynoms [BvzGG+01].

3.2 Composition of Efficient Nested BSP Algorithms

The results in this section are based on work done for my diploma thesis [Bon02]. The actual running time of implementations of parallel algorithms depends on two groups of parameters, namely "software" parameters, for example, the number of memory accesses or the degree of a broadcast tree, and "hardware" parameters, for instance, the time necessary to set up a communication or the number of available processors. This leads to the observation that for different parameter constellations different "plain" algorithms are the fastest ones. Furthermore, sophisticated parallel algorithms often introduce subproblems or make recursive calls. For example, efficient parallel algorithms for minimum spanning tree computation use parallel sorting algorithms, broadcast methods, and make recursive calls. For each subproblem, one can choose between different algorithms.

Hence, in order to have efficient parallel programs for an actual machine, they have to be *configured*. This means that one has to decide which parallel algorithm has to be taken and which subroutines have to be used on what portion of the parallel machine depending on hardware parameters as well as on the input instance for the algorithm.

For the BSP model and its extensions, a large variety of efficient parallel algorithms for many problems has been developed and quite accurately analyzed (see Section 3.1).

This enables the programmer to choose from a pool of available algorithms for the composition of the final program on his or her parallel machine. However, configuring a parallel program becomes quickly very complex due to the various parameters and (possibly mutually influencing) dependencies. Therefore, it should be done automatically by a program we call *configurator* [BMW02].

3.2.1 The Configurator: Input/Output Specification and Algorithm

In what follows, we define how to describe algorithms and introduce the term *schedule* for a given problem. This is the necessary adaptation of the BSP model. A schedule fixes the algorithms and all free parameters to be used to solve a problem and all occurring subproblems. The input of the configurator is the problem Π, the BSP computer (i. e., which BSP parameters apply), and a specification of the input of Π. The configurator works on a library of algorithm descriptions and outputs the schedule. This schedule is used during the execution to determine the real (sub-)program that will be executed. It is the equivalent to the actual parallel program.

Algorithm Description. An *algorithm description A* consists of the following five components:

BinTreeBroadcast:= $(Broadcast, p_{count}, v_{count}, t, s_{count}, s)$

- $p_{count}(n,p,p_{max}) := \{p\}$, use all processors
- $v_{count}(n,p,c) := 1$, only one variant
- $t(n,p,c,v) := \lceil \log_2 p \rceil (2 \cdot \max(n, B(p)) \cdot g(p) + L(p))$,
  one superstep per tree level, n bytes data send to each of the 2 children
- $s_{count}(n,p,c,v) := 0$, no subproblems
- s undefined function

Figure 3.2: Example for an Algorithm Description for BINARYTREE Broadcast

1. The problem Π that the algorithm solves.

2. The set p_{count} of feasible machine sizes, i. e., the machine sizes for which the algorithm A can work depending on the input for A.

3. The number v_{count} of different possible choices for fixing the free algorithmic parameters of A. For example, in a broadcast algorithm this might be the number of feasible tree degrees.

4. The function t that computes, for a given variant of the algorithm, the running time of A without the time A will spend in subcalls. Note that in the algorithm description, it is not known in advance which algorithms for the subproblems will be used in the schedule.

5. The function s that computes a list of subproblems for all possible variants of A. Each list item contains the subproblem, the description of the input of the subproblem, and the processors involved.

Figure 3.3 shows an example for the Broadcast algorithm TREE. A more formal specification of the components and a more detailed description can be found in [Bon02]. We say that A *solves* problem Π.

Schedule. Given a set $\mathcal{P}$ of problems and a set $\mathcal{A}$ of algorithm descriptions, a *valid schedule* S for a problem $\Pi \in \mathcal{P}$ and its input description is defined recursively. An algorithm $A \in \mathcal{A}$ solving $\Pi \in \mathcal{P}$ is fixed, as well as all free parameters, and there are valid schedules for all occurring subproblems $\Pi'_i \in \mathcal{P}$. The recursion terminates when there are

Let $d_v \geq 2$ be the v-th integer number, s. t. $d_v | p$ (d_v is the tree degree of variant v).
TreeBroadcast:= $(Broadcast, p_{count}, v_{count}, t, s_{count}, s)$

- $p_{count}(n, p, p_{max}) := \{p\}$, use all processors
- $v_{count}(n, p, c) := |\{d \in \{2, \ldots, p\}; d|p\}|$,
  variants are all possible tree degrees d, d is valid tree degree $\Leftrightarrow$ d divides p
- $t(n, p, c, v) := \max(n, B(p)) \cdot (d-1) \cdot g(p) + L(p)$,
  time for the first level of the tree
- $s_{count}(n, p, c, v) := \begin{cases} 0 & \text{if } p = d_v \\ d_v & \text{otherwise} \end{cases}$
  If the degree d_v is smaller than p, there is one subproblem (Broadcast)
- $s(n, p, c, j, v) := (Broadcast, n, p/d_v, 1, d_v)$,
  input for subproblem Broadcast has input size n for p/d_v processors, and is executed d_v times in parallel

Figure 3.3: Example for an Algorithm Description for TREE Broadcast

no further subproblems. So a valid schedule S can be viewed as a *schedule tree* directed from the root to the leaves.

Let S be a valid schedule for a problem Π that has to be executed on a p processor BSP machine given by its machine parameters. The *cost* of S, i. e., its predicted running time on the BSP machine, is defined along the schedule tree. The cost of the root is the running time of algorithm A (without the time spend in solving subproblems) given by the function t (see point (4) above) plus the sum of the cost of all children of the root.

Configurator. We have implemented a prototypical configurator that computes a schedule tree (in a bottom-up way) and, hence, a valid schedule with minimum cost by a brute force search testing all possible valid schedules. Note that this computation is offline, i. e., it is done only once before the schedule is used for many same-sized inputs. The configurator works for arbitrary problems Π and algorithm descriptions A. Although the execution time of the configurator may be exponential in the number of algorithms, for our MST experiments the configuration was finished in less than five minutes.

3.2.2 An Example: Parallel Minimum Spanning Tree Algorithms

As example for a complex problem with a rich combinatorial structure, a usually irregular communication pattern, and a variety of sophisticated algorithms that in turn use clever

subalgorithms, we use the problem of computing a minimum spanning tree (MST) for an undirected weighted graph. For the algorithmic background of the MST problem, see [CLR90, Chap. 24].

Minimal Spanning Tree (MST). Given an undirected graph $G = (V,E)$ with edge weights $w : E \to \mathbb{Q}$ a tree $T = (V,F)$ with $F \subseteq E$ and $|F| = |V| - 1$ is called a *spanning tree*. The weight $w(T)$ of a spanning tree is $w(T) := \sum_{e \in F} w(e)$. T is a *minimal spanning tree*, if and only if $w(T) \leq w(T')$ for all spanning trees T' of G.

3.2.3 Algorithms for the MST problem

Each of our implemented algorithms is based on the following three basic operations:

1. Algorithm KRUSKAL: Kruskal's sequential algorithm ([CLR90, Sec. 24.2]) tests for every edge, in order of increasing weights, whether it can be included in the minimum spanning tree, i. e., whether it connects two connected components (called *supervertices*) created by the edges chosen so far.

2. Operation "Borůvka step": In a Borůvka step (for a nice and detailed description, see [Göt98]) each supervertex selects its cheapest outgoing edge. These edges are added to the MST edges, avoiding cycles (by construction, these cycles are not longer than 2). After that, the new supervertices, i. e., the connected components, are calculated, the edges are relabeled according to the new supervertices, and all edges belonging to the same supervertex are removed. This step reduces the number of vertices at least by a factor of two.

   We have implemented two different algorithms for this problem, namely DENSE-BORUVKASTEP which is step (2) of algorithm MST-DENSE in [ADJ+98], and BORUVKASTEP which is in essence from [DG98, Göt98].

   DENSEBORUVKASTEP is specially designed for dense graphs. It calculates the lightest edge of all locally stored edges and then executes a parallel prefix operation to determine the edges with global minimum weights, for every supervertex.

   BORUVKASTEP creates adjacency lists for all vertices by grouping edges of the same vertices by integer sorting. Then the minima of each group are calculated by a parallel segmented prefix operation (see [Göt98], Section 4.1.4). Our implementation uses a sequential algorithm for computing the connected components.

3. Operation "MSTMERGE": This operation (Step (2) of MSTMERGE in [ADJ+98]) merges local MSTs. It uses a d-ary communication tree. d is a free parameter to be set by the configurator. Each tree node sends its MST to its predecessor, the

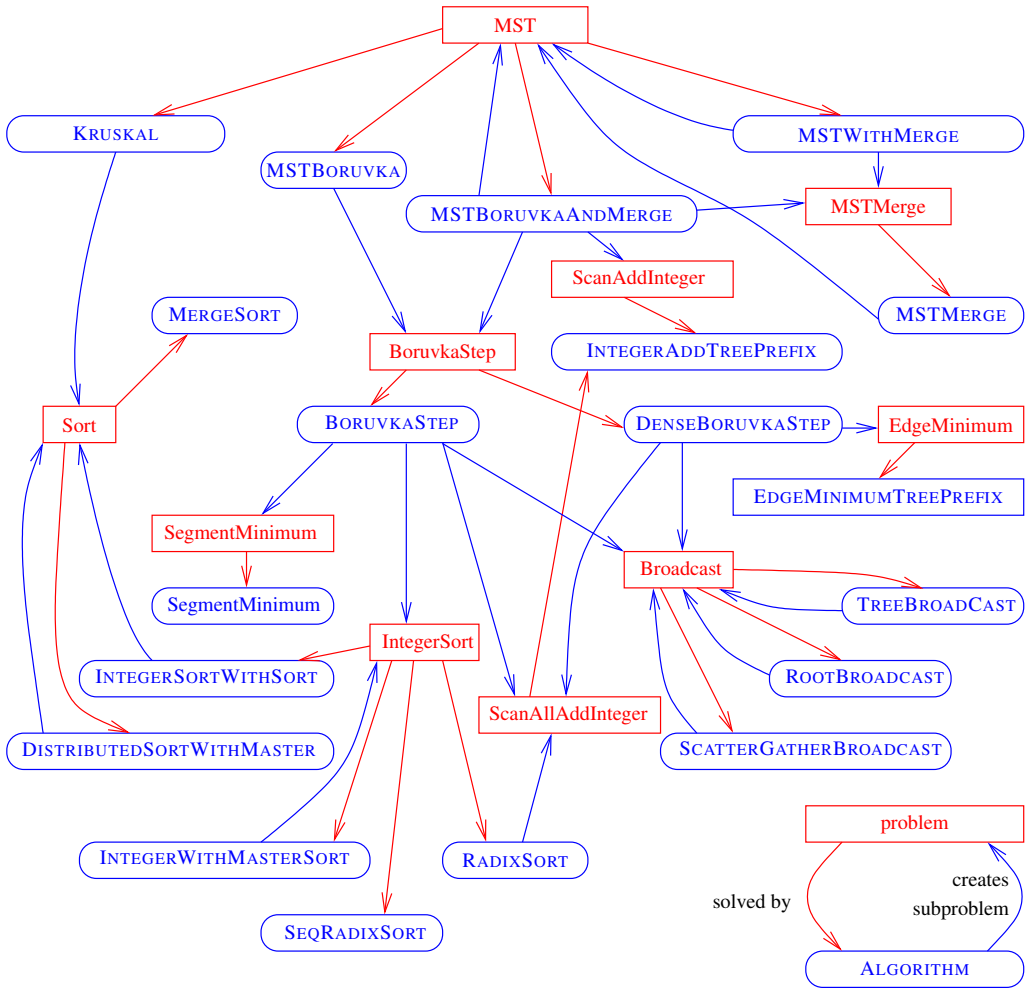


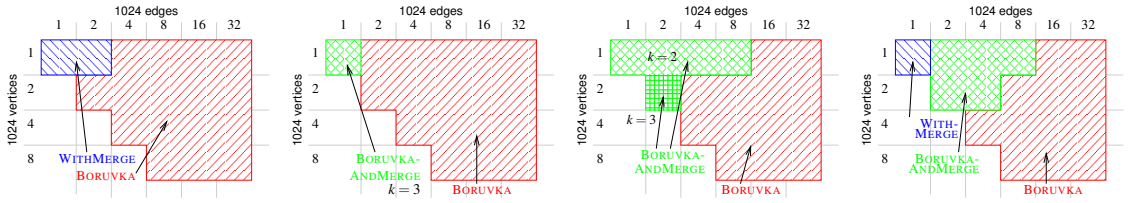
Figure 3.4: Map of Algorithms used for Configurator Test

predecessors merge the MSTs by calculating an MST of all edges received. In the end, the root of the communication tree knows the global minimum spanning tree.

Figure 3.4 shows a full map of all used algorithms, the subproblems, and their relationships. Note that the map even contains cycles that are caused by recursive calls. Of course, the configuration terminates because the parameters usually decrease.

In the following, we give short remarks on the three parallel MST algorithms that can be found in the map. The algorithmic description of our general solution of the MST problem has three main variants: MSTWITHMERGE, MSTBORUVKA, and MSTBORUVKAANDMERGE.

MSTWITHMERGE: This algorithm solves the MST problem by calculating the local



(a) $p = 2$ (b) $p = 4$ (c) $p = 8$ (d) $p = 16$

Figure 3.5: Results of the Configuration on a Pentium III Workstation Cluster with SCI

spanning tree and merges all these trees with operation MSTMERGE. Then the result is distributed from the root to all nodes.

MSTBORUVKA: This algorithm executes Borůvka steps until the number of supervertices is 1. Since each step reduces the number of supervertices by at least a factor of 2, at most $\lceil \log_2 n \rceil$ of these steps have to be executed (n denotes the number of vertices).

MSTBORUVKAANDMERGE: This algorithm combines Borůvka steps and merging. At the beginning it executes some number k of Borůvka steps in order to reduce the number of vertices, and then it calculates the minimum spanning tree using MSTMERGE. The number k of Borůvka steps is a free parameter of the algorithm and has to be set by the configurator.

3.2.4 Experimental Evaluation of the MST Implementations

Configured Programs: the Schedules

The output of the configurator is a schedule that defines for the problem and each occurred subproblem which algorithm on which processors should be used. Figure 3.6 shows an example of a schedule for the minimal spanning tree problem on a graph with 4096 vertices and 32768 edges. Horizontal there are the processors, whereas the vertical direction marks the time. Each box denotes an algorithm and the needed resources.

Figure 3.5 shows as the result of the configurator the selected algorithms and the fixing of the free parameters for different sizes of the input graphs and different number of processors for the parallel machine interconnected as a 2-dimensional torus of SCI links.

On a computer with a fast network, the algorithm MSTWITHMERGE is chosen on small graphs only. This algorithm has a small number of supersteps, namely $\log_d p$, but calculates a minimum spanning tree sequentially on $(n-1)d$ edges for n vertices in every node of the d-ary communication tree. Also, most of the processors $(p - p/d^{i-1})$ are idle

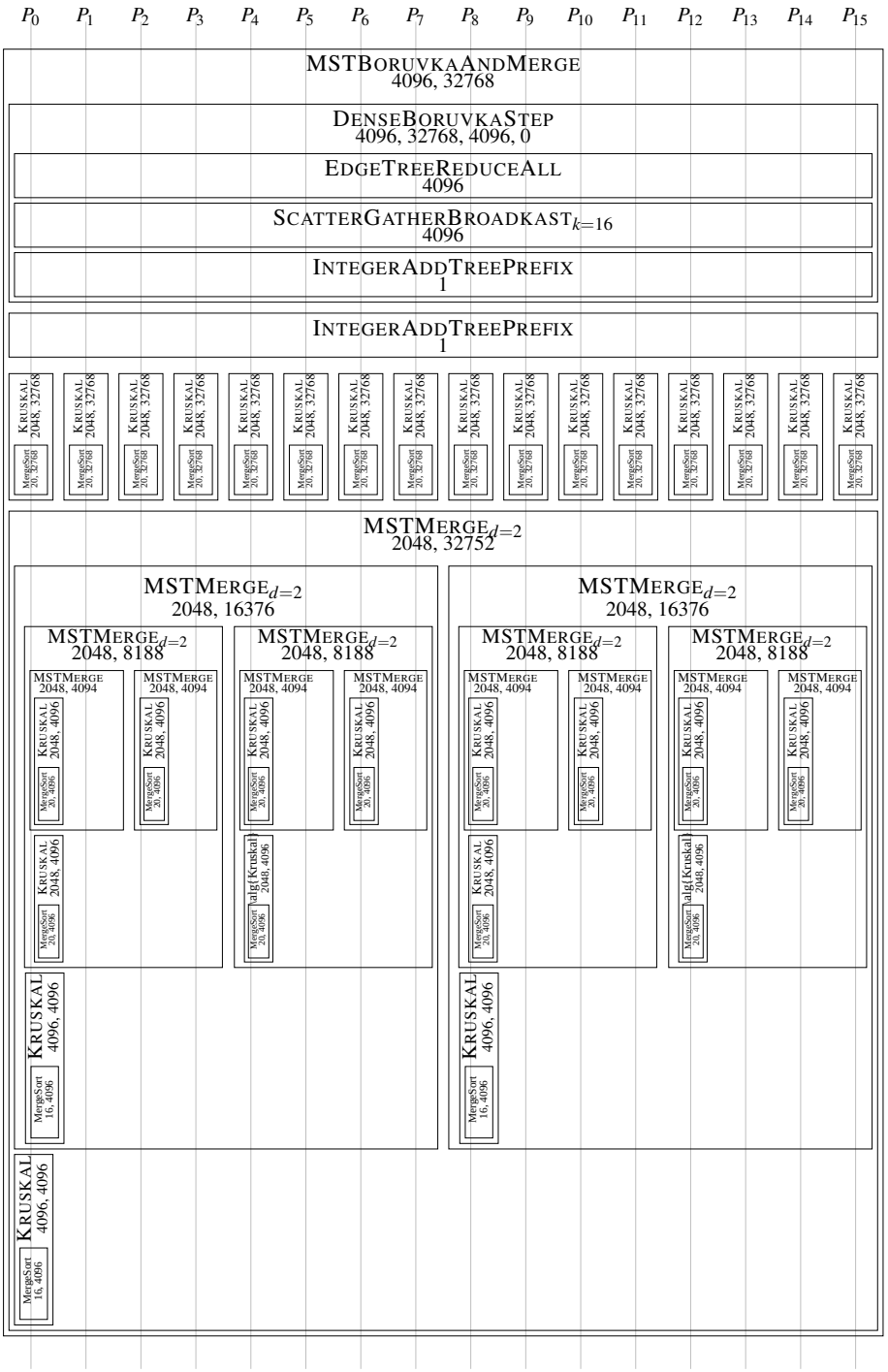
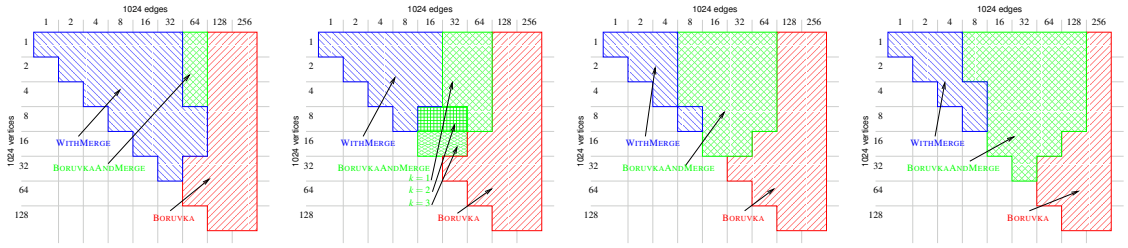


Figure 3.6: Schedule for the Minimal Spanning Tree Problem of a Graph with 4096 Vertices, 32768 Edges, for a 16 Processor BSP Computer



(a) $p = 2$ (b) $p = 4$ (c) $p = 8$ (d) $p = 16$

Figure 3.7: Results of the Configuration on a Pentium III Workstation Cluster with Fast Ethernet

in round i. If the network is slow (as it is the case with Fast Ethernet), the change from one chosen algorithm to another occurs for larger graph sizes, as can be seen in Figure 3.7 that presents the results of the automatic configuration for the cluster with Fast Ethernet.

An evaluation of the schedules, i. e., measurements of the running time of configured algorithms, and more details of the used computers and their BSP parameters are presented in Section 4.2.1 in the next chapter.

Chapter 4

Implementations

The BSP model was proposed as a bridging model between the hardware and the software, i. e., it should support the development of portable algorithms and lead to efficient implementation. In the previous chapter we gave an overview of the algorithms, in this chapter we look at implementations. We start with programming environments which support the development of BSP algorithms, including our contribution, the PUB library, and give an overview of implementations of BSP algorithm in the second part.

4.1 BSP Environments

There are several environments to support the implementation of BSP programs, developed by various people with different objectives. To improve the compatibility, a small set of core functions is defined as a standard called "*BSP Worldwide Standard*" [GHL+96], and some libraries implement these functions.

The libraries were implemented with different motivations: some focus on small, simple, and portable code (e. g., BSPonMPI), some have adapted implementations for varied parallel architectures (e. g., Oxford, PUB library); others allow to use the internet for distributed BSP calculations (e. g., Bayanihan, PUBWCL) or have a total different aim, for example, Virtual BSP for debugging BSP programs.

In the following we give a short overview of libraries (see Table 4.1 for a list) for bulk synchronous parallel computing. We focus on libraries for high performance computing on systems with a high performance network and describe our implementation, the Paderborn University BSP library, in more detail.

4.1.1 Related Work

In this section we describe four libraries for bulk synchronous parallel computing: Oxford BSP Toolset, BSPonMPI, BSPK, and GreenBSP.

Table 4.1: Overview of BSP Environments

Implementation	Language	Focus	Reference
Oxford BSP Toolset	C, C++, Fortran		[HMS+98]
BSPonMPI	C, C++	MPI based, portable	[Sui07]
GreenBSP	C, C++	basic functions	[GLR+99]
PUB library	C, C++	performance, functionalities	[BJvR03]
BSPk	C, C++		[FH96]
JBSP	Java		[GLC01]
Bayanihan	Java	master-worker with central server	[Sar98]
PUBWCL	Java	Web-Computing	[BGM06]
Dynamic BSP		Grid Computing	[MT04]
InteGrade Grid BSP	C	middleware	[GGHK05]
NestStep	Java, NestStep	programming language	[Keß00]
BSMLlib	Objective CAML	functional BSP	[HL02]
Virtual BSP	simulator	debugging	[MW00]

Oxford BSP Toolset

The Oxford BSP Toolset [HMS+98, Hil98] consists of implementations of the BSPlib standard for various parallel architectures and some profiling tools. It was the first implementation of BSPlib and is sometimes referred to as Oxford BSPlib. It supports remote memory access and message passing. Focussing on these basic functions there are only 20 primitives, all other features, including broadcasting and parallel prefix operations, are implemented in a first level library on top of BSPlib.

There are many publications that deal with this implementation. Some describe the usage [HMS+98], answer questions about BSP in general and the implementation of the toolset [SHM96], others study implementation techniques, for instance, how to implement BSP over a TCP/IP network [DHS97] or the Cray T3E interconnection network [HDS97], or extend the library by load balancing and fault tolerance [HDL98].

BSPonMPI

BSPonMPI (cf. project webpage [Sui07]) is an implementation of the BSPlib standard from Wijnand J. Suijlen under the guidance of Rob H. Bisseling, the author of the textbook about BSP [Bis04] and maintainer of the BSP worldwide website. It is easy to install (using the GNU autoconf tool) and should be runable on all POSIX architectures that supports the message passing standard MPI [MPI95]. It is optimized for MPI, so the author states it should be used wherever native implementations are not available, as it is

faster than other implementations (PUB, Oxford BSP Toolset) using MPI.

Green BSP Library

The Green BSP Library [GLR+99] was designed to be as simple and portable as possible. It can only transmit packets of fixed size (e. g., 16 bytes, can be configured at compile time), so the user has to split the messages into packets and recombine them at the destination. The only receive function `bspGetPkt` returns a pointer to one packet, i. e., to receive a large message one have to call this function quite often, and all parts have to be copied together using a lot of memory copy operations, each of small size. This is neither comfortable for the user nor very efficient for large messages.

BSPk

BSPk [FH96] is a library for message passing and distributed shared memory for BSP. The authors introduced two new aspects. First, they implement *lazy barriers*, i. e., the supersteps are not separated by global barriers, it is just guaranteed that messages sent in superstep i will be received in superstep $i+1$. If a processor will not receive a message, it does not have to wait for other processors at the end of a superstep. To use this optimization, the user has to inform the library about all expected incoming messages (the same type of synchronization is implemented in our PUB library and called `bsp_oblsync` there). This is formalized in the Oblivious BSP model (Section 2.2.3).

The second new idea is called *Communicable Memory* by the authors. They try to tackle the problem of buffering and copying. If user data is sent, it normally has to be packed in some kind of frame, for instance, a header containing its size and sender. So the data is copied into a buffer after the header. If the communication network is fast (compared to the processor and memory speed), the copying could decrease the communication bandwidth significantly, for example, a remote memory copy on the Cray T3E computer is faster than a local copy, due to the limited memory bandwidth. In BSPk the library can manage the memory, so it can be allocated it in proper frames. For the same reason the PUB library can allocate memory for a message (`bsp_createmsg`).

Furthermore, BSPk was the first library that supports message passing and distributed shared memory, other earlier implementations supported only one type of communication.

4.1.2 Paderborn University BSP (PUB) library

In this section we will introduce our Paderborn University BSP (PUB) library [BJvR03, pub08]. It was developed to support high performance implementations of BSP programs for monolithic parallel computers, and later extended to workstation clusters. The most important design goal was efficiency, so we focus on avoiding buffering and copying

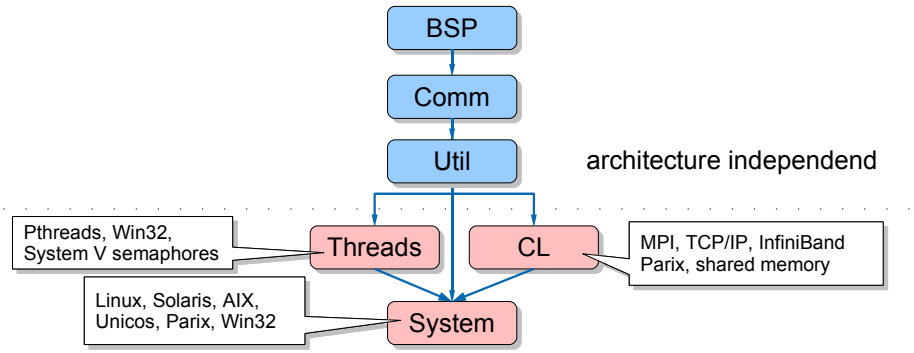


Figure 4.1: Modules of PUB

wherever possible. Furthermore, we keep in mind software engineering techniques. Although developed using the language C, PUB is object oriented, i. e., there are objects (e. g., messages), and methods for using them. PUB is available for many different architectures, i. e., different hardware (e. g., CPUs, network devices) and various software (operating system, communication libraries). To achieve this portability, PUB consists of different modules (Figure 4.1), including one module for the operating system (System) and one for the communication (CL).

Although developed for BSP implementations, PUB offers much more functionalities. The BSP module is just one use of the more flexible communications functions. More details about the BSP part of PUB can be found in [Rie00].

Basic BSP Functionalities

There are two different groups of functions for communication: message passing and direct remote memory access. These two kinds of communication are widely used in all parallel system, for example, the *Message Passing Interface* (MPI [MPI95]) is a common standard for message passing, whereas the Shmem library from Cray supports remote memory accesses. As PUB is a BSP library, all these functions have a bulk-synchronous semantic: messages sent in one superstep are received in the next synchronization and remote memory operations are also performed at the synchronization. Besides normal barrier synchronization PUB supports the *oblivious synchronization* of the Oblivious BSP model (cf. Section 2.2.3), where the user has to specify the number of received messages and the synchronization does not need any communication at all.

Collective Communication Operations

Several global operations are provided: Broadcast and multicast algorithms, which automatically adapt their parameters like block size and tree degree to the hardware archi-

tecture, parallel reduce and scan operations with arbitrary associative and commutative operators.

BSP Objects

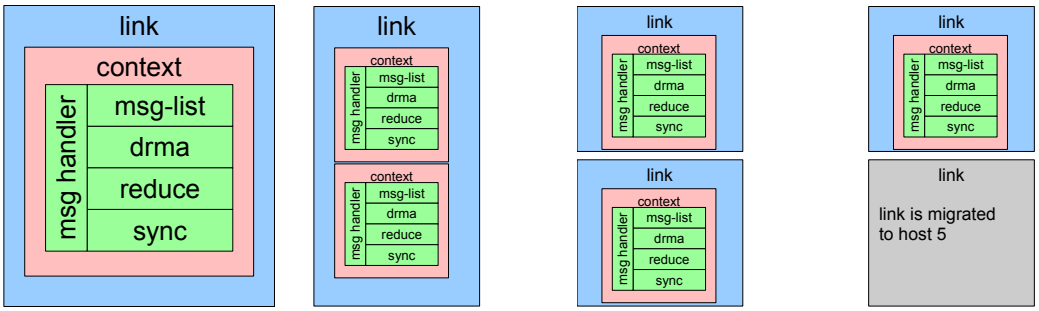
Another feature is the possibility to dynamically partition the processors into independent subsets. After a partition operation, each subsystem acts as an autonomous BSP computer, i. e., subsequent barrier synchronizations involve only the processors in the subsystem. Thus, the PUB library supports nested parallelism and subset synchronization. This feature is especially useful if different subalgorithms have to be performed independently in parallel. Without the possibility to create subsystems, the algorithms have to be interleaved, which complicates the implementation. Furthermore, the implementation might be inefficient if the algorithms have different synchronization requirements.

With these BSP objects it is also possible to create totally independent BSP computers, which can be used in different threads for executing more than one parallel algorithm at once. Virtual processors (used for load balancing on heterogenous workstation clusters, Chapter 6) are also realized with BSP objects.

Communication Layer of PUB

The communication functionality of PUB is based on *active messages*. This is similar to event based system, for example, graphical user interfaces. Each received message triggers the execution of a message handler. To describe the message processing in more detail, we look at two communication objects: commlinks and commcontexts. A *commcontext* is a kind of virtualization of the parallel message passing system. In each context messages can be sent and received independently of messages in other contexts. This is especially useful when calling sub programs and is the basis for the BSP partition functionality, where a context covers only a part of the processors. The creation of such a context should be very fast, in particular it should not need any communication at all. Each context has an identifier used to route messages to it. Every processor needs to know all identifiers of the contexts on all other hosts. As we do not want to communicate, these identifiers are the same on all hosts. This means that the order of creating and destroying contexts have to be the same on all participating processors.

To support really independent communication, which can even be used simultaneously in different threads, PUB uses *commlinks*. Commlinks are similar to commcontexts, but can be used independently of each other. This freedom leads to a performance drawback: the creation of a commlink needs a global total exchange communication. Commlinks can not be used standalone, but they are the containers for the commcontexts, i. e., a commlink can contain one or more commcontexts, whereas each commcontext belongs to exactly one commlink. Commlinks are also used to implement virtual processors, virtual memory, and process migration (see Section 6.3).



(a) normal (b) bsp_dup (c) bsp_threadsafe_dup (d) bsp_migratable_dup with migrated processor

Figure 4.2: Comm Objects in PUB

Each message contains three identifiers in its header: one for the link, one for the context, and one for the message handler. When a message is received, the receive thread calls the message handler of the link, which calls the message handler of the context, which executes the right message handler of the BSP layer. If a context or message handler is not known yet, the message is buffered in a queue until the destination is registered. This occurs, for example, in a BSP program, if a message of the next superstep is received.

The library creates one commlink with one commcontext at startup. The functions `bsp_dup` and `bsp_partion` create additional contexts, `bsp_threadsafe_dup`, `bsp_exclusive_dup`, and `bsp_migratable_dup` use extra links. There are several message handlers registered by PUB in each context: one for normal messages, one for remote memory operations, one for global operations like reduce, and one for synchronization messages. Figure 4.2 shows the objects for (a) normal BSP programs, (b) with BSP subgroups, (c) with independent BSP objects, and (d) with two virtual processors (one has left the host and is executed on the host with id 5).

The message handler for BSP messages is registered every superstep, i. e., it will get a new identifier in each round, so messages from different supersteps are not mixed.

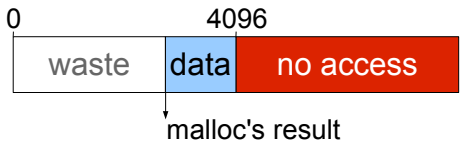
Debugging

In general it is a hard task to find mistakes in implementations, thus a lot of tools were developed to support the debugging, for example, debugger, which allows to set breakpoints, step through a program step by step, and watch the values of variables. Most of these tools support sequential programs only, some can handle programs using threads, similar to the PRAM model, but there are only a few tools that support watching a parallel program (e. g., TotalView). Thus a standard approach to find errors is to put a lot of debug output into the program by hand. The PUB library supports different techniques for debugging.

Automatic Execution of a Debugger. If a fault occurs, for instance, a wrong memory access (segmentation fault) or an assertion failed, PUB can automatically start a debugger and attach it to the affected process. This feature is known from Microsoft's developer tools, there you are asked if you want to debug a program if an access violation occurs. This is especially usefull for parallel programs running on a workstation cluster, because it is an annoying job to find the host where the faulty process is being executed by hand.

Pointer Type Check. To each structure of PUB, for example, a message, an additional field *id* is added, which stores the size of the structure. Each time a pointer to a structure is given, the library checks this id. Thus pointers to wrong memory positions or not initialized data structures are detected quickly.

Range Check. One class of errors are illegal memory accesses, i. e., a program writes to a memory cell with an invalid address. Too small memory allocations are one reason for this. Thus we support a kind of range checking of memory regions allocated by the function `malloc`. The memory is organized in pages of size 4096 bytes for x86 processors. Each time the processor wants to access a memory cell, the virtual address of the cell is translated into a real memory page and an offset inside the page by the Memory Management Unit (MMU) inside the processor. The MMU checks, if there is physical memory mapped to this address, and if it is accessable by the process, i. e., one can have different access flags (read, write, execute) attached to memory pages. If there is no memory, or the access is not allowed, an exception occurs and the operation system is called, which can swap in the memory, or generate a segmentation fault signal or similar. We use this feature to find accesses to the memory direct after an allocated block. We will explain this by a simple example. If the program allocates 512 bytes, we allocate 2 memory pages of size 4096 each, and return a pointer to the offset 4096-512 inside this block. Furthermore we change the access permissions of the last page to *no access*.



All memory allocations and freeings can be monitored, such that memory leaks can be detected.

Message CRC check. To verify the communication system, PUB can calculate a checksum for all messages and check, if they are transmitted correctly. We use the standard Cyclic Redundancy Check (CRC) [PW72] algorithm for the checksum.

Table 4.2: Comparison of Oxford BSP Toolset and PUB Library

	Oxford BSP Toolset	**PUB Library**
Point-To-Point Messages	✓	✓
Direct Remote Memory op.	✓	✓
High performance op.	✓	✓
Oblivious synchronization	–	✓
Broadcast/multicast	library on top of BSP	automatically optimized for the machine
Collective communication Operations	library on top of BSP	✓
Subgroups	–	Partitioning, independent groups
Thread-safety	–	on most architectures
Virtual Processors	–	✓
Fault tolerance	extension [HDL98], source not available	Checkpointing of virtual processors
Load balancing for workstation clusters	extension [HDL98], source not available	Processmigration, various load balancing algorithms, easy to extend
Asynchronous message passing	–	possible, active messages model
Debugging tools	–	execution of debugger, pointer type checking, range checking, memory leak detection, message CRC checking
Profiling	call-graph profiling tool [HJSV98], BSP Pro [ZKX99]	show idle and compute times, size of messages for each superstep

Comparison with other BSP Libraries

Comparing with other implementations, our PUB libraries offers much more functionalities as any other system. The support for decomposable BSP, oblivious synchronization, support for virtual processors and memory, and fault tolerance are some of its features. For a comparison with the Oxford BSP Toolset see Table 4.2.

Furthermore, the possibility to use asynchronous communication is very useful for some algorithms, which does not fit well into the BSP model. Using this functionality, one losses the advantages of BSP (e. g., simplicity, easy cost model), but for some applications

Table 4.3: Implementations of BSP algorithms

Problem	Environment	Platform	Reference
Inner product	BSPlib		[Bis04, Bis07]
LU decomposition	BSPlib, MPI	Cray T3E	[Bis04, Bis07]
FFT	BSPlib, MPI	SGI Origin 3800	[Bis04, Bis07]
Matrix multiplication	Green BSP	SGI Challenge, NEC Cenju, Linux workstation cluster	[GLR+99]
Sparse-matrix-vector multiplication	BSPlib, MPI	Linux workstation cluster	[Bis04, Bis07]
Ocean simulation	Green BSP	SGI Challenge, NEC Cenju, Linux workstation cluster	[GLR+99]
N-Body simulation using Barnes-Hut	Green BSP	SGI Challenge, NEC Cenju, Linux workstation cluster	[GLR+99]
Minimal spanning tree	Green BSP	SGI Challenge, NEC Cenju, Linux workstation cluster	[GLR+99]
	PUB	Linux workstation cluster, SCI and Ethernet	[BMW02]
	PUB	Parsytec results on CC-48	[DG98]
Shortest paths	Green BSP	SGI Challenge, NEC Cenju, Linux workstation cluster	[GLR+99]
Multiple shortest paths	Green BSP	SGI Challenge, NEC Cenju, Linux workstation cluster	[GLR+99]
Volume rendering		SGI PowerChallenge, IBM SP/2, workstation cluster	[XL97]

this seems to be necessary.

4.2 Implementations of BSP Algorithms

For the success of parallel algorithms in general, and the BSP model in particular, it is quite important that there are easy to use implementations of the algorithms. The main reason for the success of Java is the availability of large, powerful class libraries with implementations of algorithms for many standard problems.

There are some implementations in the bulk synchronous settings, but most algorithms listed in Chapter 3 are analyzed only theoretically. Furthermore, there is no a central instance providing a list or a library of implementations. Rob Bisseling, maintainer of the BSP worldwide website [BSP07], has collected a list of researchers working in the BSP

area, but since now there is no list of software or anything similar. Table 4.3 contains examples of BSP implementations of algorithms. It shows the solved problems, the used toolkits or libraries, the BSP computers for the evaluation, and the references.

4.2.1 Combining BSP Algorithms

Often implementations of algorithms by different providers cannot be combined to one program because of missing standards for BSP implementations. There is the BSP worldwide standard, which defines communication functions for BSP programs, but many implementations use other libraries or are implemented directly on top of simple message passing libraries like MPI. Another non trivial part is the specification, including the format and the distribution of the input, or the number of supported processors.

It is crucial for an efficient combining of different algorithms that the data is in the appropriate format and does not need to be rearranged. For example, sometimes the input of a subproblem is stored in a non consecutive way in the memory, but the subalgorithm needs it as one memory block, thus one have to copy all data before calling the subalgorithm. In many applications these memory copying leads to inefficient algorithms.

Furthermore, an efficient way of calling subalgorithms is needed, for instance, the partition feature of the PUB library. Using this technique, it is possible to separate the communication of the subalgorithms, and to run several subalgorithms in parallel. The BSP worldwide standard does not support such functionality.

A proposal of a system to build libraries with BSP algorithm can be found in Section 3.2.

Evaluation

We have implemented all algorithms of the map in Figure 3.4 on page 30 to evaluate the schedules made by the configurator. First, we want to know if the running time of the algorithm is predicted accurate enough to find the optimal schedule, furthermore, we are evaluating the performance of the resulting algorithm.

The Used Parallel Machines. For the experiments, we used a cluster of 96 Linux workstations with two different communication mechanisms. The cluster was operated by the Paderborn Center for Parallel Computer (PC^2). Each node is a Dual-Pentium III workstation operating at 850 MHz and includes 512 MB memory. For the communication, there are two alternatives: One can use a 2-dimensional torus of SCI links as interconnection network, or Fast Ethernet with a Cisco Catalyst 5509 switch (and, hence, a complete graph as interconnection network). For the communication, MPI and TCP/IP are used, respectively.

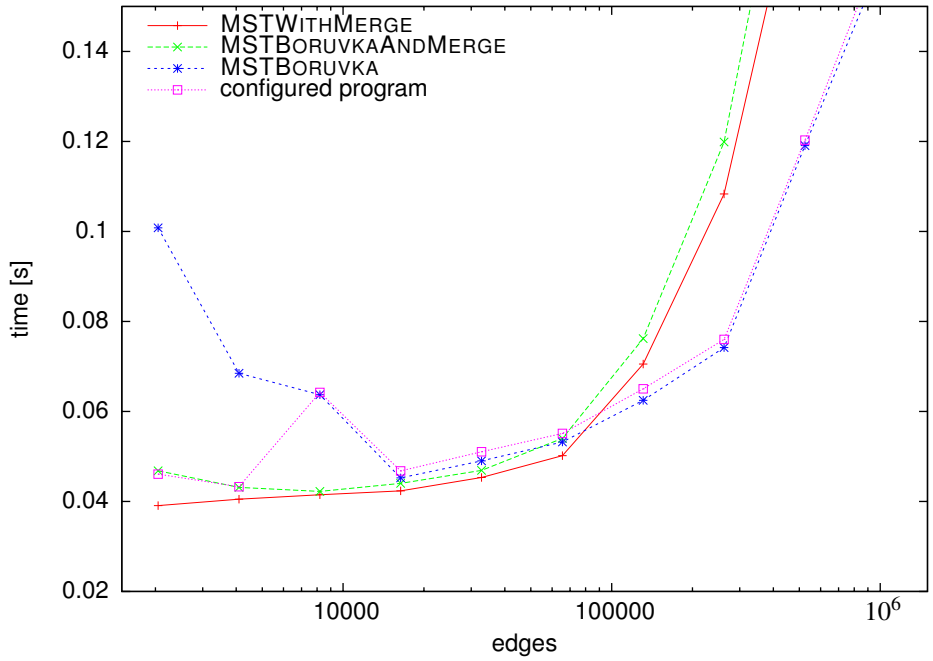


Figure 4.3: Running Time, Random Graphs, 2048 Vertices, 16 Processors, SCI

Table 4.4 presents the BSP* parameters of these two communication systems of the parallel computer. They were used by our configurator for the prediction of the cost of our implemented algorithms. A compilation of the BSP* parameters of further machines (e. g., Cray T3E, Parsytec CC) can be found in [Rie00].

Measured Running Times. In the following we will use the parallel minimum spanning tree problem as an example and present the results of two significant series of measurements, one for each communication mechanism. More measurements can be found in [Bon02].

Table 4.4: The BSP* Parameters of the Pentium III Workstation Cluster

(a) MPI with SCI (2D torus)

p	B [B]	L [µs]	g [ns^{-1}]
2	316	2.10	32.4
4	509	4.49	37.5
8	389	6.28	43.9
16	21	9.08	194.5

(b) TCP/IP with Fast Ethernet (complete graph)

p	B [B]	L [µs]	g [ns^{-1}]
2	3092	98.9	117.1
4	1467	145.5	121.8
8	264	140.7	629.2
16	239	152.7	617.9

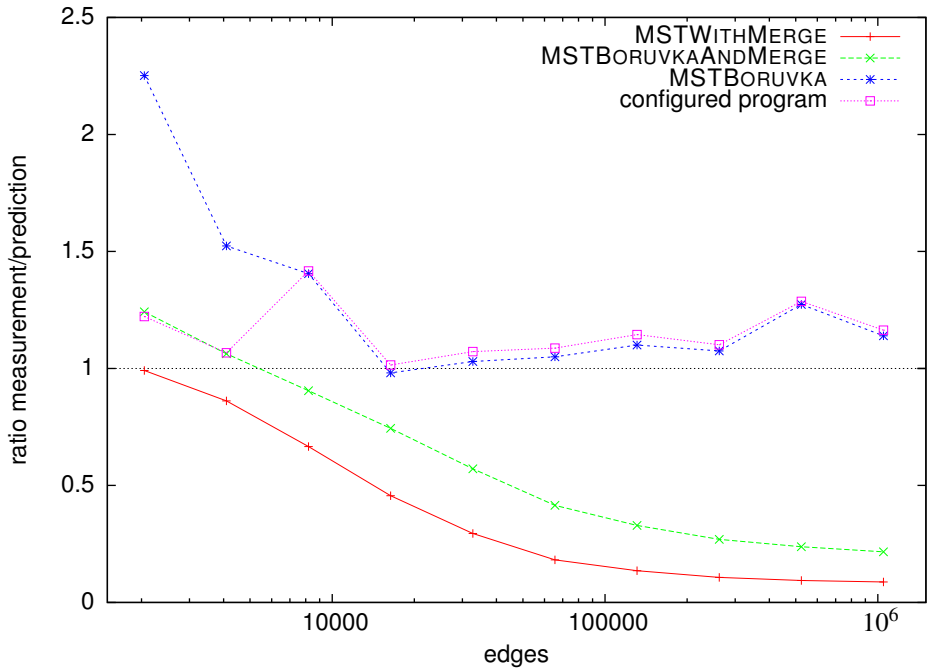
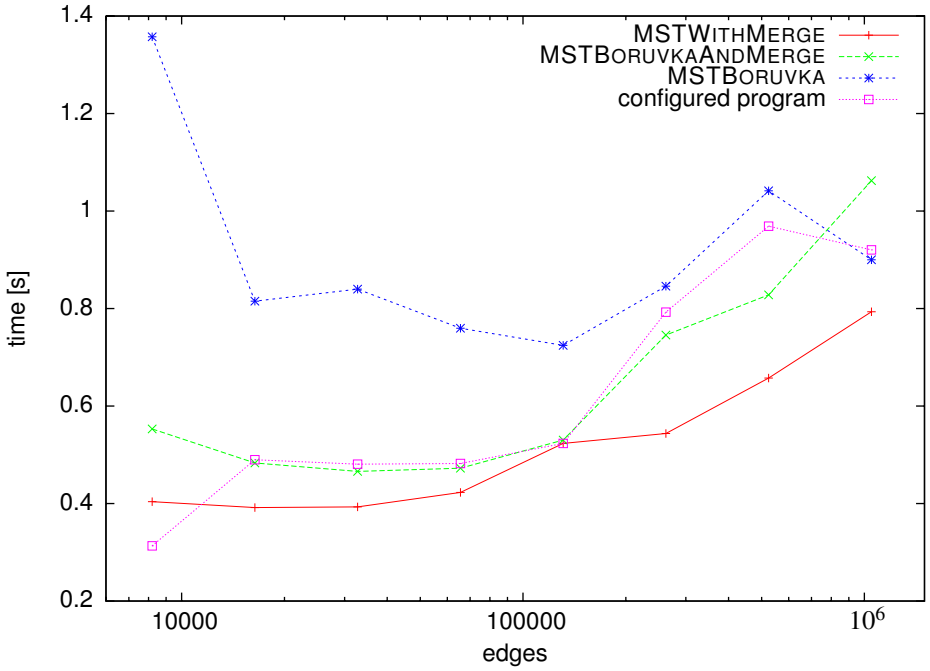


Figure 4.4: Accuracy of the Predictions, 2048 Vertices, 16 Processors, SCI

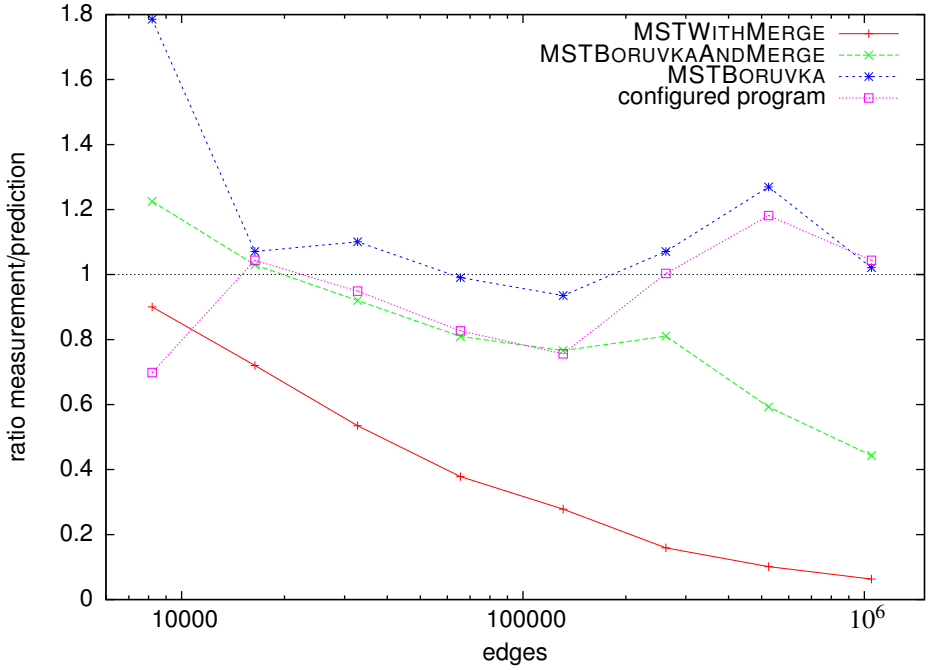
Figure 4.3 shows the results on the workstation cluster with $p = 16$ processors. Our graphs consist of $n = 2048$ vertices and randomly chosen edges. Figure 4.3 shows the running times, Figure 4.4 the ratio of the measurements and the BSP cost, i. e., the predicted times, for the SCI case. The divergence for the algorithms that use the operation MSTMERGE is due to inaccurate predictions of the local work. In algorithms based on merging, the sequentially merging of minimum spanning trees dominates the running time. The model counts the local memory accesses of these operations. However, the algorithms run much faster than predicted due to cache effects.

The configured program results, as Figure 4.3 shows the best running time, if the input graphs are dense. Otherwise, it comes close to the fastest algorithm. The peaks that can be observed for the configured program in both parts of the figure at 8192 edges is due to the fact that the configurator prefers MSTBORUVKA to MSTBORUVKAANDMERGE too early.

Figure 4.5 presents the measurements and the prediction accuracy ratio for the Fast Ethernet communication on $p = 16$ processors and graphs with $n = 8192$ vertices. The configurator does not choose the best variant which is mostly MSTWITHMERGE because using Fast Ethernet the network load being a dominating parameter in the running time is not sufficiently covered by BSP. More specifically, if the input graph is dense, during a broadcast only one processor sends data, all other processors receive data. So there



(a) running time



(b) accuracy of the predictions

Figure 4.5: Results for Random Graphs with 8192 Vertices on 16 Processors, TCP/IP

is no high network load, and the gap g, listed in Table 4.4 and measured under high load, is inappropriately large. So the running time is considerably overestimated by the configurator.

Concluding Remarks. In this section, we have evaluated the configuration approach to obtain fast parallel BSP programs for solving algorithmically complex problems. As a case study, we have presented the results of applying the configuration approach to the minimum spanning tree problem.

The configured algorithm performs well in practice, although the predictions of the running time is sometimes quite inaccurate. The main reason for this is the fact that the algorithm description has to provide information about subproblems, depending only on the description of the input. In our example, we are given the number of vertices and the number of edges in the graph, and we have to predict the number of supervertices created by executing a Borůvka step, which is half the number of vertices in the worst case, but normally much smaller. So the prediction gives an upper bound on the execution time, but is not always accurate enough to compare the average running time of algorithms.

In [Bon02], a case study for the broadcast problem can be found. For broadcasting the running time does not depend on the input, thus the predictions are more accurate.

Chapter 5

BSP for Parallel System on Chip

The bulk synchronous parallel model was designed to be a bridging model between the development of algorithms and the hardware. In this chapter we study how BSP can help to develop efficient parallel architectures by designing a parallel system on a single chip. Furthermore, this shows that BSP is also feasible for on-chip parallelism, which has some special characteristics, for instance, high bandwidth and small memory sizes.

Some of the work has been done in the project GigaNetIC[1]. We will start with a short introduction of the project and continue to present related work in the area of parallel systems on a single chip. The main part of the chapter deals with the architecture and the evaluation of our system.

5.1 Project GigaNetIC

The *GigaNetIC* project aimed to develop high-speed components for networking applications based on massively parallel architectures [BSR+03]. A central part of this project is the design, evaluation, and realization of a parameterizable parallel processing unit. A particular attraction of the GigaNetIC project grounded on the interdisciplinary coupling of the different research groups that marked the project right from the beginning.

There are members of three research groups involved. We from the Research Group of Algorithms and Complexity develop and analyze models for the architecture and the on-chip interconnection network. The Research Group of Programming Languages and Compilers develops a parallelizing compiler, and the Research Group of System and Circuit Technology is intrusted with the realization of a resource-efficient system design and with the development of hardware accelerators for special embedded applications to achieve a higher throughput and to reduce energy consumption. As an industrial partner,

[1] GigaNetIC was founded by the Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung)

Infineon Technologies provides state of the art chip production technologies needed for such a complex System-on-Chip (SoC) design.

5.2 Related Work

There are different projects that tries to utilize the huge amount of transistors that can be integrated into a modern chip. The first approach was increasing the sizes of the caches and the number of execution units (instruction level parallelism, execution of more than one instruction per clock cycle). This leads to very huge and complicated processors like Intel's Itanium2 (including up to 24 MB cache). But to improve the performance significantly, one needs more processor cores. One possibility is to integrate standard (huge) processors cores into a single chip, examples are the Power4 processor (IBM, first dual core processor, 2001), Intel Core2Duo series, AMD Athlon 64 X2, Intel Itanium2 "Montecito" (2 Cores, 24 MB cache, largest processors with 1,720 million transistors), and SUN UltraSPARC T2 "Niagara" (8 cores, 64 threads).

Another approach is implementing many more smaller processor cores, for example, the Single-Chip Message-Passing Parallel Computer (SCMP) [BGB+04] developed at Virginia Tech. The system consists of a 2-D mesh of tiles, each tile contains CPU, memory, and network router. The communication of the system is based on active messages. There are thread messages and data messages. Thread messages start a thread at the destination, data messages contain an address and store the data at that address (direct remote memory access). The network connections on the edge of the chip are used for input and output.

Detailed models of the system were being developed for VHDL (Very High Speed Integrated Circuit Hardware Description Language) and SystemC (C++ library for system descriptions), and there seems to be a simulator for the system, because the authors provide several benchmark data: Floyd-Warshall all pairs-shortest-path (transitive closure of the adjacency matrix), iterative conjugate gradient sparse-matrix solver, and the neighborhood stressmark which estimates the gray-level co-occurrence matrix energy and entropy by calculating histograms of the sums and differences of pairs of pixels in an image.

The Cell processor (Cell Broadband Engine Architecture, developed by Sony, IBM, and Toshiba [KDH+05]) is a kind of mixture of both ideas. It consists of one standard processor core (*Power Processing Engine*, PPE), eight fully-functional co-processors (*Synergistic Processing Elements*, SPE), and a specialized high-bandwidth circular data bus connecting PPE and SPEs. The PPE runs the operation system, has control over the SPEs, and can start, stop, interrupt, and schedule processes running on the SPEs. The SPEs are not fully autonomous and require the PPE to do any useful work. They work as co-processors to support the main core in certain applications, for example, decoding/encoding MPEG streams, generating or transforming three-dimensional data, or undertaking Fourier analysis of data. The chip was designed for the video game console PlayStation 3, but there

are other system using Cell available as well, including blade servers.

Due to the heterogeneous architecture of the system, it is not a a straightforward task to develop algorithms for Cell. Ohara et al. introduced an MPI microtask model [OIS+06], based on the message passing standard MPI. The programs have to be partitioned into a collection of small microtasks that fit into the local memory of the SPEs. These microtasks are scheduled statically to get an implementation for the streaming model of the Cell.

5.3 Design of the Architecture

In the first section we will describe the architecture of our parallel system on chip. We will start with an overview of the components, specify the communication protocol, and provide more details on the parts in the last subsection.

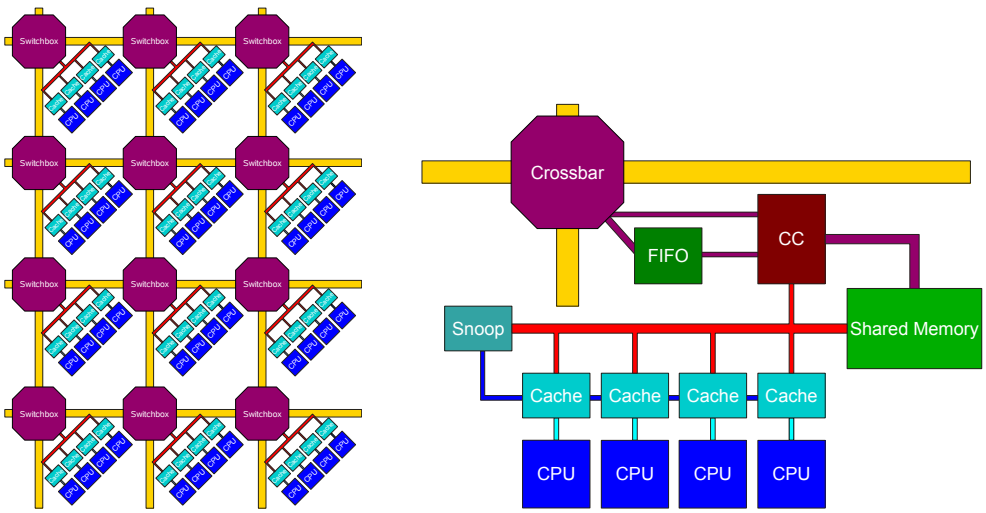
5.3.1 Overview

Looking at existing parallel computers, there are two different architectures: shared memory and distributed memory. A global shared memory is often considered to be very comfortable for the users, as they can use threads to design parallel algorithms. The implementation of shared memory is costly, because a memory bank can only be accessed by a small constant number of processors at the same time. So the performance of these architectures does not scale with the number of processors, or a big and thus expensive crossbar is needed, which connects the processors to a huge number of memory banks.

Another approach is distributed shared memory, which leads to *Non-Uniform Memory Architectures (NUMA)*. In NUMA, for example, SGI's Altix, the access time to the memory varies, for instance, some part of the memory may be local and thus faster.

Distributed memory systems are widely used, all cluster systems built of standard components and a high speed network are of this kind, and most machines at the top of the Top500 list of supercomputers [Str06] are distributed memory architectures. In our architecture we try to combine the advantages of shared and distributed memory. We use small clusters of processors including shared memory connected by a switched bus system. These clusters are connected by a fast on chip interconnection network with routers and communication controllers. The controllers have the capability to send messages autonomously from one cluster to another without slowing down the computation of the processors. The part used for the communication of a cluster is called *Switchbox*.

Figure 5.1(a) shows an overview of the architecture, the blue boxes are the processors with local caches, the shared memory and the communication controllers are inside the Switchbox. The configuration ($3 \times 4 \times 4$) shown in the picture is just an example and can be adjusted to the needs.



(a) mesh of clusters (b) single processor cluster

Figure 5.1: Architecture

5.3.2 Communication Protocol

The communication on the chip between clusters of processors is based on message passing. To avoid the problem that a large message can block a lot of small ones, we do not allocate the whole path from the source to the destination of a message as done in wormhole routing. Instead, messages are split into very small parts called *flits*. Each flit is transmitted to the next Switchbox in one cycle using parallel wires. To recreate the message, each flit contains – in addition to the routing information – a *flit id* and a *flow id*. The flit id is the number of the flit inside the message; the flow id is used to distinguish between messages. The first flit of a message is marked as a packet header and contains the size of the message. This is used to allocate enough buffer space for the flits, and to notice when the last flit has been received.

The routing is *oblivious*, i. e., the path for a packet is dependent only on its source and destination. Each flit is sent to the next switchbox again and again, until an acknowledgment is received. Waiting for these acknowledgments leads to a reordering problem. As we want to send a flit every cycle, we cannot wait for the acknowledgment, so we use a ring buffer and try to send a flit every cycle. This can change the order of the flits, for instance, if the first flit cannot be sent, the second flit is tried before the next try of the first one. As the head of a message must arrive first, we do not allow a data flit to overtake a header flit. Furthermore, we do not allow the other direction, a header flit cannot pass a data flit, too. This is needed because we must avoid that messages from the same sender are mixed.

5.3.3 Components

In Figure 5.1 (b) the components of a single cluster are shown in more detail. The Switch-box contains a crossbar with buffers, the communication controller (CC), which handles the message passing, the shared memory, and the snooping-slave (Snoop) for the cache control.

CPU. As a central processor unit we use S-Core, a RISC processor core that is binary compatible with Motorola's M-Core M200 architecture [Mot98a, Mot98b]. The processor has been developed at the University of Paderborn by the Department of System and Circuit Technology as a soft core using the hardware description language VHDL. This gives us the opportunity to reuse the core for different target technologies. The S-Core is a 32 bit RISC two-address machine with a straightforward load/store architecture. It has two banks of sixteen 32-bit registers, which can be alternatively used in user mode. Each instruction has a fixed length of 16 bits. This results in a high code density and therefore reduces memory demands for the application code. Beyond that, each instruction, except for the load and store instructions, works only on the registers. The execution of most instructions takes one clock cycle. The S-Core features a short three-stage pipeline and an addressing interface that supports byte granular access.

The architecture can easily be expanded by adding application-specific instructions or coprocessors to the core, examples include application specific optimizations for packet processing for a network processor [NPPR07, NPSR05]. Our implementation of the CPU is resource-efficient and delivers reasonable performance for embedded systems. In a 0.13 µm 1.2 V Infineon standard cell process the S-Core needs less than 0.25 mm^2, and clock frequencies of more than 200 MHz can be achieved in this technology.

We use this processor as a demonstrator, because it is available and we are free to modify it as we want, for example, implement new instructions for the synchronization. As the processor is very simple and does not have a floating point unit, it is not suitable for high performance scientific computing. But it can be replaced by other, more complex processing elements. In the following, we will focus on the communication architecture, which is useful for all kind of parallel on chip systems and is independent of the type of processors.

Cache System. Each processor owns a small cache for instructions and data to reduce the congestion at the shared memory. The caches are organized with the *MOESI* cache coherence protocol [AMD06]. MOESI supports the following states of a cache line:

Modified: Cache line holds the most recent copy of the data, data in memory is incorrect, and no other cache contains a copy.

Owned: Cache line holds the most recent copy like in state modified, but other caches may also have copies. Only one cache may own the data, all other caches having

the same data have state shared. The owner is responsible for writing the data back to the memory.

Exclusive: Cache contains the only copy of the data, and the data in the memory is up to date.

Shared: Cache line holds the most recent copy, but other cache may have copies as well. The data in the memory is correct, or one other cache contains the data and has state owned.

Invalid: Cache line does not contain valid data.

As the bus connecting the caches with the shared memory (RAM) is a switched one (ARM Advanced Microcontroller Bus Architecture (AMBA) with a Multi-Layer-AHB-Interconnect-Matrix [ARM99, ARM01]), the caches are not able to monitor the bus by itself. Thus, we use a *snooping slave* to control the caches. Each write access of a processor that changes a cache line state is noticed to the snooping slave and broadcasted to all other caches. Also, in case of cache miss, the snooping slave is asked if the data is stored in any other cache. This is needed to get the newest copy, and also reduces the needed memory bandwidth, as the data is fetched from another cache if possible. Details on the hardware implementation can be found in [NLPR07].

Communication Controller. The *communication controller* (CC) manages the communication of a processor cluster with the world outside. It is able so send and receive messages autonomously. The CC uses memory mapped input/output (I/O), i. e., it is addressable by the normal memory bus. It supports asynchronous sending, blocking receiving, and probing for messages or acknowledgments. In the following we present the process of sending a message to another cluster.

Sending. To send a message, a processor executes a write access to the bus. Different from normal I/O-registers, the send functionality does not use a single address, but a larger range of the address space. The address bits are used to transmit additional information to the CC in a single write operation on the processor bus. The start address of the data is transmitted by the data, the information about the destination and the length is encoded in the address of the write access. The following data is encoded in the address:

Bits	31–28	27–25	24–21	20–17	16–6	5–0
Data	fixed to 0111_2	z	x	y	*size*	*id*

z is the output port at the destination cluster, x, y are the relative coordinates of the destination cluster, *size* is the size of the data (in flits), *id* a user defined identifier.

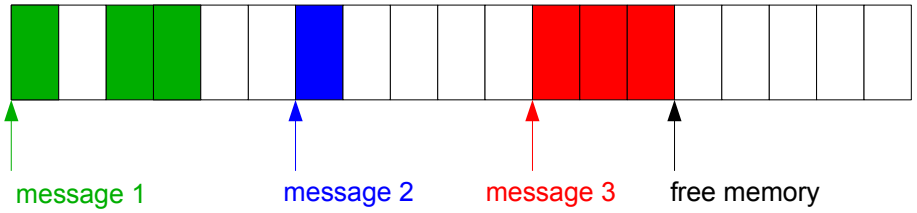


Figure 5.2: Receive Buffer

The communication controller stores the send request in an outgoing queue. This queue is processed asynchronously. Until it is empty, the CC dequeues one request, reads the data from the shared memory, and sends it through the on-chip network. As the memory is a dual port memory, memory accesses of processors and communication controller can be executed concurrently in our design.

Receiving. Incoming flits are combined to messages by the communication controller. It uses a part of the shared memory as a ring buffer for received messages in the following way. One registers stores the address of the free memory. For each header flit introducing a new message, the suitable space in the memory is reserved by advancing the free-memory-pointer. Next the start of the packet is stored in another register. For each following data flit, the controller searches these registers for the corresponding message and stores the data of the flit at the right memory position. Thus, the communication controller needs one register for each simultaneously receiving message. Our implementation uses at least as many registers as we have clusters in the system, so we can guarantee that we will always find a free register if a new message arrives.

Figure 5.2 shows an example of a receive buffer with three incoming messages, message 3 is completely received, for the other messages some parts are missing. In message 1 two flits have overtaken the second flit which is still missing.

To receive a message, each processor can read from a special memory address. The result is either the address of a received message in the shared memory or zero if there is no message available and the address for the non blocking mode was used. One can also wait for either a message or an acknowledgment.

A message should be freed as soon as possible to allow the reuse of the buffer space for other incoming messages. If there is not enough space in the buffer, incoming header flits are rejected (i. e., they are not acknowledged, so the other node will try so resend it every second cycle) and the network could be overflowed by blocked flits, especially since no other flit is allowed to overtake the header flit.

More details on the communication network and its implementation in hardware is given in [PNPR07].

Table 5.1: Performance of Simulators

Simulator	System Size	Application	Sim. Speed
Cluster-Simulator[2]	1 S-Core	Dhrystone	7 231 kHz
Cluster-Simulator[2]	4 S-Core	Dhrystone	2 290 kHz
SBArray-Sim[2]	1 Cluster, 1 S-Cores	Dhrystone	315 kHz
SBArray-Sim[2]	1 Cluster, 4 S-Cores	Dhrystone	173 kHz
SBArray-Sim, Amba Matrix[2]	1 Cluster, 4 S-Cores	Dhrystone	167 kHz
SBArray-Sim, Amba Matrix[2]	4×4 Cluster, 64 S-Cores	Dhrystone	10 kHz
SBArray-Sim, Amba Matrix[2]	4×4 Cluster, 64 S-Cores	Total Exch.	11 kHz
VHDL simulation[3]	2×4 Cluster, 32 S-Cores		100 Hz
FPGA implem.		independent	20 MHz

[2]Intel Core2 CPU 6700, 2.66 GHz
[3]Pentium IV, Hyperthreading, 3 GHz

5.4 Toolchain

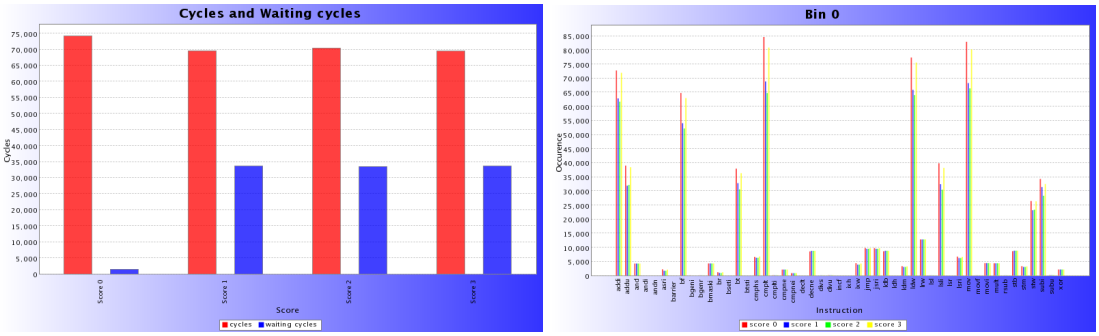
This section shows the toolchain for the parallel system on chip. To ease the software development and testing for the system, we have developed a couple of tools: cross compiler, shell scripts for compiling and linking, standard libraries, communication libraries, and several simulators.

5.4.1 Simulators

For the evaluation of the architecture a couple of simulators have been developed. They differ in the parts of the system they can simulate, the needed resources (computer, FPGA), and the accuracy. Tabular 5.1 gives an overview over the simulators and their performance, the software simulators are described in more details in the following.

The fastest simulator simulates a cluster of up to four processors with shared memory and a simple multiprocessor cache. It is very fast and gives a rough impression of the computation performance of the system. But as it does not simulate the bus system and the caches in whole details, some effects like bus congestion cannot be examined. The instruction decoder of the simulator is generated automatically from a processor description in UPSLA (unified processor specification language [KLST04]), so it is very useful to evaluate changes to the instruction set of the processor quickly. The simulator outputs a lot of statistics, for instance, cycles, wait cycles, occurrences of instructions, and pairs of instructions that can are candidates for creating a new superinstruction, for example a combined load-and-xor (Figure 5.3).

The second simulator provides a cycle accurate simulation of the whole system. It uses SystemC, a C++ library to describe and simulate concurrent processes. SystemC



(a) cycles and waiting cycles (b) occurrences of instructions

Figure 5.3: Visualization of the Profiling Output of the Cluster Simulator

```
width 2
height 2
number_of_pes 4
0 0 0xd0000000 dhrystone.startup
0 0 0xd000002c dhrystone.0
0 0 0xd01050dc dhrystone.1
0 0 0xd020a18c dhrystone.2
0 0 0xd030f23c dhrystone.3
stack 0x008242e0 0x008342e0 0x008442e0 0x008542e0
comm_buffer 0xd08642e0 0xd08742df
cache_size 16384
line_size 16
associativity 2
mem 0xd0000000 0xd08742df
debug dhrystone.debug
ambamatrix yes
```

Figure 5.4: Configuration File for SystemC based Simulator SBArray-Sim

was originally developed by Synopsys, Inc., an electronic design automation company, but later (1999) turned into an open source project [sys07].In 2005, IEEE approves the IEEE 1666 -2005 standard for SystemC [IEE05]. The simulator is based on the source code from Norman Nagel for his diploma thesis [Nag04], but extended to simulate our architecture more precisely (communication, bus system, caches), and optimized.

SystemC provides C++ classes and templates for modules and connections (ports and signals) between them. It is possible to specify action methods and event handlers, which are called if a port state changed. In our implementation, components (e. g., CPU, memory, and cache) are modules, but also busses, the arithmetic logic unit (ALU), and containers (e. g., CPU-cluster) are realized as SystemC modules. SystemC contains a scheduler which executes all action methods and calls the appropriate event handlers. The simulation must not be dependent on the order of the execution, events that happen at the same time are processed sequentially in arbitrary order. Although the simulated system is parallel, the free SystemC implementation provided by the SystemC community [sys07] does not support parallel computers and does not profit from multi core architectures, for instance, the Intel Core2Duo used for our benchmarks.

The simulator is configured by a configuration file. An example is shown in Figure 5.4. This file contains the size of the system (width×height×number_of_pes), the memory map, i. e., files mapped into the memory (code, initialized data segments), ad-

```
 1 SIGINT: cancel simulation
 2 Simulation time: 41273
 3 Cluster (1,1)
 4 communication controller
 5   input: recv 0, count 0, ptr 0, used 0
 6   output: write 0, count 0, ptr 0, used 0
 7 ncore 0
 8 PC: 0x00000648, opcode ?
 9 PC-history 0x000005fa 0x00000290 0x00000286 0x000006ee 0x000006de
10 Backtrace: strcmp Proc_8 main _main
11 REG[00]: 0x00824258 REG[01]: 0x00000008 REG[02]: 0x00004fd0 REG[03]: 0x000028a0
12 REG[04]: 0x00000009 REG[05]: 0x00002f00 REG[06]: 0x00000640 REG[07]: 0x00005068
13 REG[08]: 0x00004fb8 REG[09]: 0x00005098 REG[10]: 0x00000020 REG[11]: 0x00004fbd
14 REG[12]: 0x00002ee0 REG[13]: 0x00000003 REG[14]: 0x00004ff0 REG[15]: 0x00000020
15 memory dump of stack (from 0xd0824258)
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
17 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
18 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
19 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                        [...]
```

Figure 5.5: Debug-Output canceling a Simulation using CTRL-c

dresses of stacks and communication buffer, and parameters for the stack and the bus system. Furthermore, a file with symbol information for debugging can be specified. Using this information, the simulator can output function names instead of just addresses, in case of an error, or if the user interrupts the simulation. The output for one cluster (communication controller) and one of the processors is shown in Figure 5.5. The symbols are used for the backtrace (functions on the stack) output. The PC-history output gives the last five addresses where the program counter changes by a branch instruction. The configuration file is generated by the compile script scc, which is described in Section 5.4.2.

5.4.2 Compiler

For the comfortable development of software, we want to program the system using a high level programming language instead of assembler. As the used processor core (S-Core) is compatible with Motorola's M-Core processor (with some additional features), we can use the standard GNU C compiler[4], to generate the object files. For access of our extension of the CPU (e. g., new instructions), we use inline assembler.

For the generation of the code for the parallel system, we provide a compile script, which calls the compiler several times and generates the object files for each processor with adapted addresses. It also hides the differences of the targets (simulators, FPGA) and generates the appropriate output format (e. g., plain binaries, s-rec-files). It creates a short startup code to set the stack pointer and to jump to the right code. This is needed, because all processors will start the execution after the reset signal at the same address

[4]version pre 4.0.0 from the cvs repository in January, 2005, later versions contain a bug and fail to compile some files

```
scc [architecture args] [additional compiler args]
    --help                    this text
    --target=<target>
        clustersim            simulator WG Kastens
        sbarraysim            systemc simulator
        wbcluster             wishbone VHDL cluster
        wbcluster-fpga        wishbone FPGA cluster
        rcos                  wishbone cluster without shared memory
        ambacluster           ambacluster with cache
    --output-format=<format>
    --width=<w> --height=<h>  size of the array
    --no-pes=<n>              number of PEs per switchbox
    --pcscore                 use pcscorec instead of gcc
    --pcscore=<ext>           use special version of pcscorec
    --stack-size=<size>       stack size in bytes
    --shared-stack            stack in shared memory (wbcluster)
    --heap-size=<size>        size of heap per PE (default 1048576)
    --shared-size=<size>      size of __sharedMem
    --mem-size=<size>         size of memory per PE (rcos, default 32k)
    --cache-size=<size>       cache size in bytes
    --line-size=<size>        size of a cache line in bytes
    --associativity=<n>       associativity of the cache
    --cache-stats             output statistics for cache use
    --verbose                 output compiler calls
```

Figure 5.6: Syntax of the Compile Script

(0x00000000). Furthermore, it produces the configuration files for the simulators and a script to start them. Thus compiling and testing is almost as easy as for a native processor.

Figure 5.6 shows the syntax of the compile script scc. The script also support the use of another compiler, pcscore, instead of gcc. This is a compiler developed by one of the project partners at the University of Paderborn, the research group of Professor Kastens. Although it does not always optimize the code as good as gcc, it has the advantage that changes at the instruction set of the processor can be realized and evaluated quite simple and fast. Another version of this compiler that can automatically generate parallel code for all processors in the cluster is being developed. Using this compiler, the user will see a cluster as a single processing unit. In our work, we would like to utilize the whole parallelism of the machine to execute parallel algorithms, so we do not use any technique to automatically generate parallel code.

5.4.3 System Libraries

We use a small standard C library with some adaptations and optimizations. The output functions (e. g., `printf`) use the architecture dependent output method, either a processor trap or writing the output to a special memory address.

As copying of memory blocks is a crucial operation for our implementation of BSP, we make some effort on optimizations of this function. The S-Core processor allows reading and writing multiple registers with a single instruction, so we save the registers, use them

Table 5.2: Library Functions for the Hardware Access

gn_get_processor_id, gn_pid, gn_pes	gives the processor switchbox, processor ids and the number of processors in a cluster
gn_write_comm_buffer, gn_read_comm_buffer	write to resp. read from communication buffer, a 32 bit shared register in each cluster
gn_barrier	barrier synchronization inside a cluster
gn_send_data	send data from shared memory to another switch box
gn_get_next	return pointer of next ack or packet
gn_wait_for_packet	wait until a packet is received and return pointer
gn_get_ack	return pointer to an acknowledged packet
gn_free_packet	free queue at place with pointer
gn_get_out_counter	return numbers of packets in send queue
gn_cache_invalidate, gn_cache_write_back, gn_cache_lock, gn_cache_unlock, gn_cache_prefetch, gn_cache_prefetch_exclusive	control of the cache (not all functions are supported on all architecture)

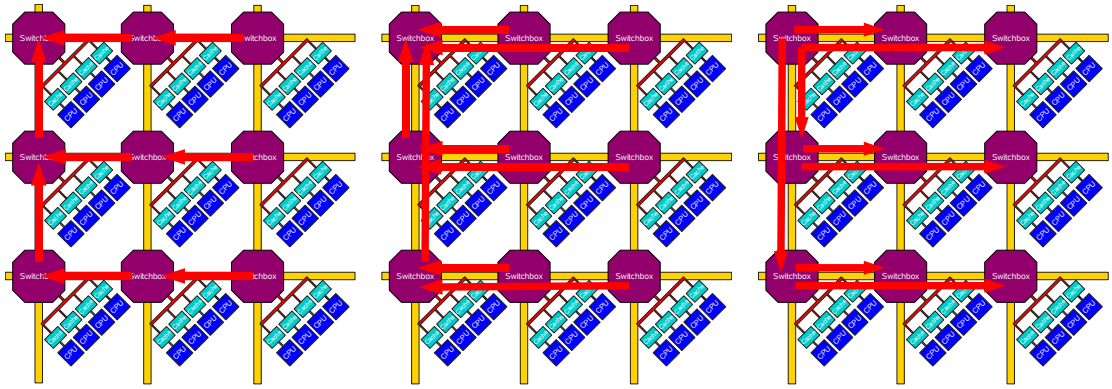
to copy the data, and restore their content afterwards. This overhead is worthwhile only for blocks bigger than a threshold, thus our implementation of the memory copy function (`memcpy`) contains several different copy loops copying 1, 8 or 11 data words at once. The following lines of S-Core assembler code show the body of the loop for copying blocks of 11 words:

```
/*
    r0 source
    r2 destination-source
    r3 destination-source - 44
    r4 length in 11 word-blocks
*/
     1 decne r4
     2 .block11CopyLoop:
     3     ldm r5-r15,(r0)
// r0 ← destination
     4     addu r0,r2
     5     stm r5-r15,(r0)
// r0 ← source
     6     subu r0,r3
     7 loopt r4, .block11CopyLoop
```

Some additional functions for accessing the hardware are also provided, see Table 5.2. An example using the communication functions for sending and receiving messages can be found in Figure 5.8 in Section 5.5.1.

5.4.4 BSP Library

For the implementation message passing in the BSP style, we have to deal with the two levels of communication. Inside the clusters, we have shared memory, thus sending of a message is just a memory copy operation, or, if we do not change the data until it is read, it is sufficient to send a pointer to the data (high performance operations).



(a) phase 1, degree 3 (b) phase 1, degree $\mathcal{O}(\sqrt{p})$ (c) phase 2

Figure 5.7: Synchronization

Shared Memory Cluster. The communication inside the cluster is realized by writing into the shared memory, similar to the PUB implementation for parallel shared memory machines like the Cray T3. PUB uses p receive queues, one for each processor, and a send is done by appending the message to the queue of the destination. For the synchronization of the accesses of different processors to the queue, an atomic operation like compare-and-swap is needed, but our architecture does not provide such operations. So we use p^2 queues, one for each sender-receiver pair.

Sending a message, i. e., a block of memory, is done in two steps: first, the data is copied into a buffer, second the address of the message in the buffer is appended to the appropriate queue. We need to copy the message to meet the BSP specification, because the data may be changed between the send and the next synchronization. For the high performance send, we do not need to copy the message and just appended a pointer to the data.

All queues exist twice. One for the messages received in the actual superstep and one for the messages of the last superstep. In the synchronization the two groups of queues are swapped (by changing a pointer to the queues to use), and the queues for receiving messages are cleared. All processors have to change the queues at the same time; otherwise messages of different supersteps could be mixed. This is realized by synchronization barriers of the processors inside a cluster. For this purpose we extend the instruction set of the processor by a barrier instruction, which can synchronize an arbitrary subgroup of processors in hardware. The execution time without idle time is only one cycle, thus, synchronization of the shared memory cluster is very fast.

On-Chip Communication. For the synchronization of the whole system we use the same technique as in the PUB library for parallel computers: each processor counts the

number of messages sent to each other processor, and in the synchronization the global sums of all these counters are computed, so every processors knows the number of its incoming messages. Next all processors wait until the number of expected messages is received. With this algorithm it is possible that processors are in different supersteps, but a processor cannot start a superstep until all other processors have at least started the synchronization procedure.

For our architecture, we do the global computation in two phases. First, the counters of the processors inside the clusters are added using shared memory and the barrier instruction for the synchronization, second these numbers are added globally. The clusters are connected by a mesh network. Each cluster, more precisely the first processor of each cluster, receives the results from the right neighbor, adds its own numbers, and sends the calculated partial sum to its left neighbor (Figure 5.7(a)). The processors of the first column receive the data from the right and the bottom and send their results to processors in the top direction. After this calculation, processor 0 of the cluster in the top left corner knows all data and broadcasts it the same way back to all the other clusters (Figure 5.7(c)).

To decrease the latency, we implement another variant of this algorithm. All processors in a row send the data direct to the first column, the processors in that column add all the numbers and send them to the top (Figure 5.7(b)). This communication uses a tree with a higher degree ($\mathcal{O}(\sqrt{p})$ for quadratic meshes) and is faster for a small number of clusters. For a larger systems a combination (fixed degree, but larger than 3) should be used.

5.5 Evaluation

We use several benchmarks for the evaluation of our architecture. In the GigaNetIC project, some low level network benchmarks have been performed, for example, IP header classification and cyclic redundancy checks (CRC).

For parallel computers and workstation clusters, we use an LU-decomposition algorithm as a benchmark for scientific calculation. The algorithm uses not only a lot of memory, it is also based on floating point operations, whereas our processor core S-Core does not have a floating point unit. In the following, we use benchmarks based on integer computations.

We did two kinds of benchmarks: first, low level communication benchmarks to determine the communication performance of the system without using BSP. Next, we use application benchmarks to find good parameters for the system (e. g., cache size) in a more realistic scenario. Furthermore, we want to compare the performance with a standard sequential processing unit.

```
1  if(pid==0){ // processor 0
2     gn_print_time(); // print start time
3     dy=(dest/4) / (int)_cluster_width;
4     dx=(dest/4) % (int)_cluster_width;
5     // send message
6     gn_send_data(dx,dy,0,id,sizeof(msg),&msg);
7     gn_get_ack(id);  // wait for acknowledgment
8     p=gn_get_next(id,0,1,1); // wait for answer
9     gn_print_time();
10    gn_free_packet(p);
11 }else{
12    if(pid==dest){
13       p=gn_get_next(id,0,1,1); // wait for packet
14       gn_free_packet(p); // free data
15       // send answer
16       gn_send_data(dx,dy,0,id,sizeof(msg),&msg);
17       gn_get_ack(id);  // wait for acknowledgment
18    }
19 }
```

Figure 5.8: Source Code Ping-Pong Test

5.5.1 Communication Benchmarks

The determination of the BSP parameters g and L is the standard benchmark for bulk synchronous parallel computing. As we consider worst case running time estimations, we stress the network as much as possible. In the BSP model, the total amount of transmitted data for a h relation does not affect the time, but for many implementations this is not accurate (cf. E-BSP, Section 2.2.4). Thus, we vary the network congestion and measure the two extremes: only one message (cheapest h-relation) and a total exchange, where each processor sends a message to each other. As we expect better performance for messages inside the clusters and communication time for a message is a function of the number of switch boxes to cross, we also measure the time needed for messages of different path lengths.

Ping-Pong. The first benchmark shows the native communication performance without using the BSP library. It shows the time needed for a small message from processor P_0 to processor P_i (ping) and the answer from P_i to P_0 (pong).

The main part of the source code for the benchmark is shown in Figure 5.8. Figure 5.9 shows the result for the Ping-Pong benchmark for two different configurations of the system: a linear array of 8 clusters and a mesh of 4×4 clusters. The time needed for ping-pong communication increases linearly with every hop, as expected. The additional cost for passing a switch box is 10–12 cycles. Surprisingly, the test is significantly faster if the

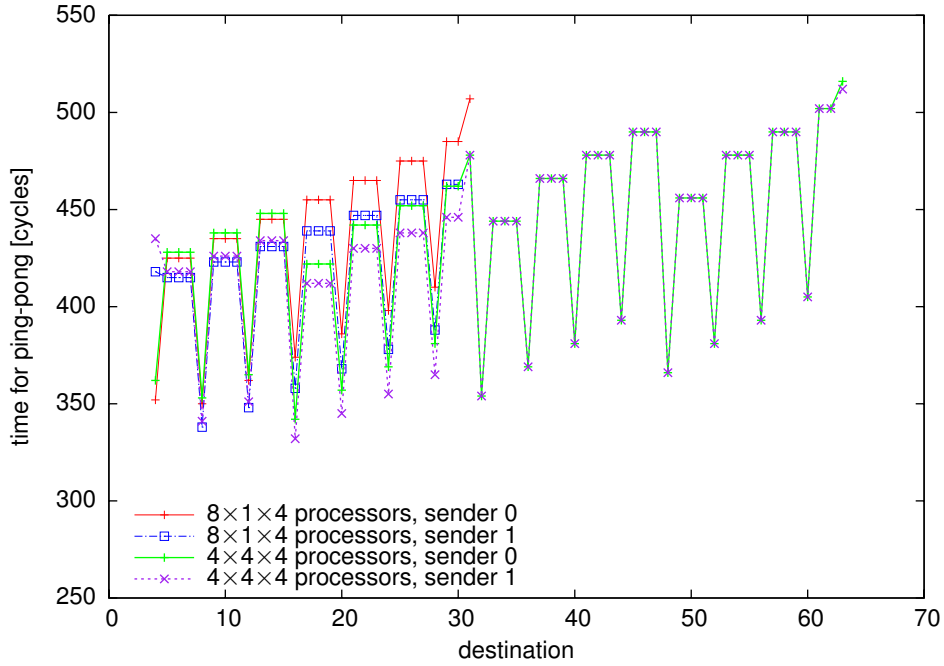


Figure 5.9: Ping-Pong Benchmark: Latency

receiver is the first processor of a cluster. We could not figure out the reason for this. We do not prioritize processor 0. All processors that are not involved in the communication wait using the barrier instruction, so they do not use any shared resource, for instance, the bus or the shared memory. Furthermore, the bus arbiter schedules the bus accesses with a round robin strategy, so no processor is preferred. The only difference between the processing elements is the following. When a cache miss occurs and the cache line is contained in more than one other cache, it is fetched from the cache with lowest id. However, this does not explain the different running times for the tests as we do not use the shared memory concurrently in this benchmark.

The same effect occurs for the sender: for small systems it seems to be faster if one but the first processor sends the message. For the larger system, the running time is not dependent on the sender processor.

The object codes for the different processors are generated in different runs of the compiler, so it could differ. For instance, code using the processor identifier, a constant known at compile time, could be optimized diverse, but the only differences in the generated assembler codes are the different addresses in the memory.

Total-Exchange. Using the BSP library for communication tests, we have two different types of costs: time needed for sending the messages and the time for the synchronization.

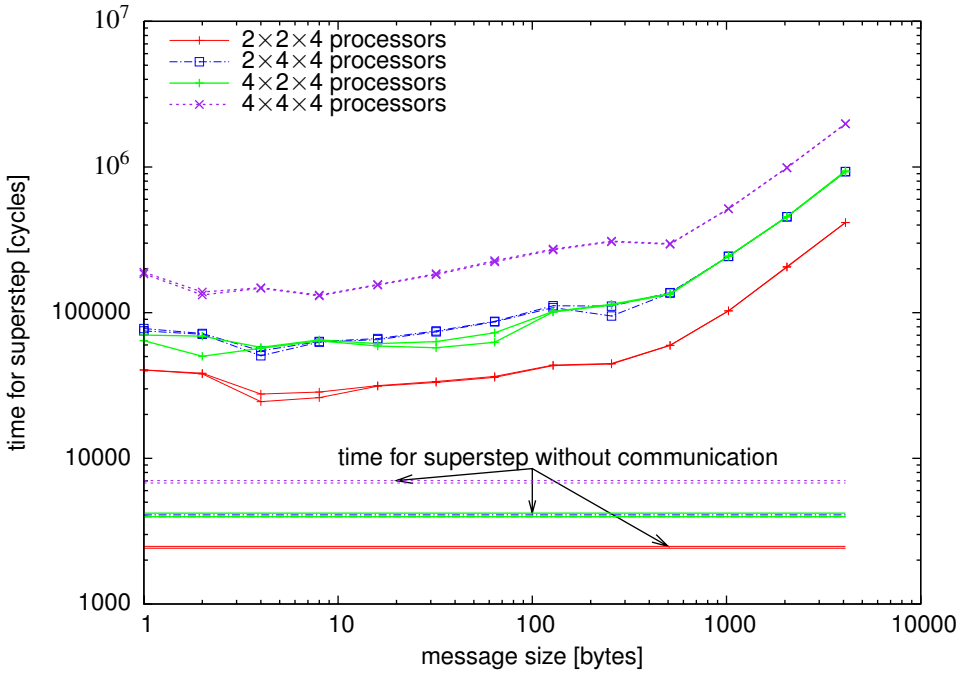


Figure 5.10: Total Exchange, Length of Supersteps

Figure 5.10 shows the length of the supersteps in clock cycles for different message and cluster sizes. Note that the amount of data each processor is sending and receiving (h in the BSP-model) is dependent on the system size, so the time should increase linearly with the number of processors if the system scales well. In the same figure, the time needed for the synchronizations without sending messages is shown (parameter L of the BSP-model). The figure shows two curves for each test, the minimal and the maximal needed cycles over all processors.

The parameter $g(p)$ corresponds to the slopes of the graphs in the figure. We got the following values for $g(p)$:

p	16	32	64
$g(p)$	6.3	7.4	7.5

Figure 5.11 shows the bandwidth per processor, which is the inverse of g. The overhead for the synchronization decreases as the message size increases, and the achievable bandwidth is almost independent of the system size for our simulated systems.

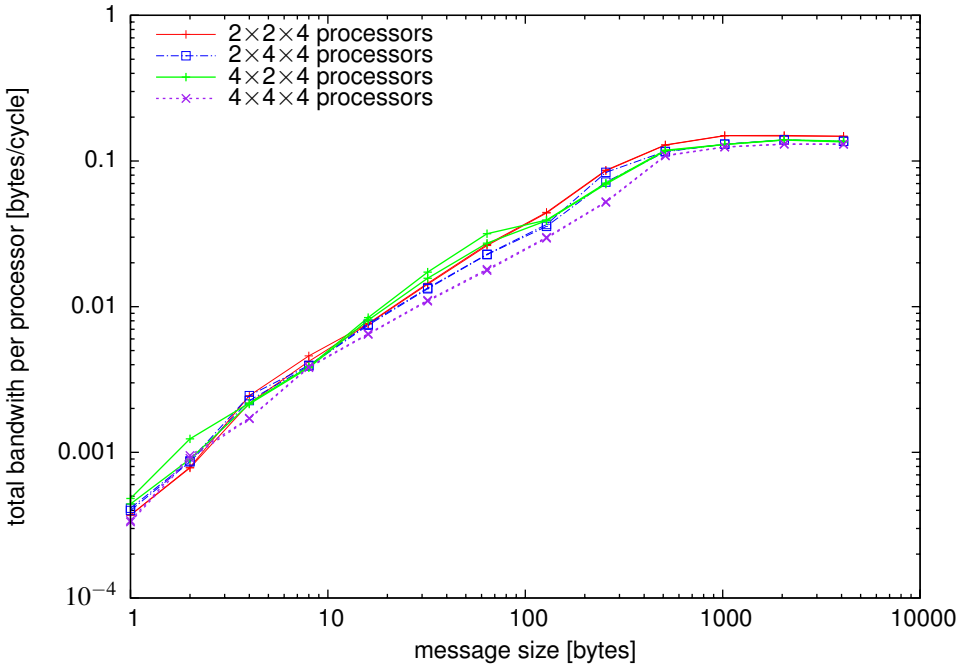


Figure 5.11: Total Exchange, Bandwidth

5.5.2 Application Benchmark

We have run two application benchmarks to evaluate the influence of the parameters of the architecture to the performance for realistic applications: Solving 3-Satisfiability and sorting.

3-Satisfiability. As first application benchmark we use a BSP implementation for the 3-Satisfiability (3-SAT) problem. Satisfiability is the problem of determining if the variables of a given boolean formula can be assigned, such that the formula is satisfied, i, e., it evaluates to true. The formula consists of a conjunction of *clauses*, each clause is a disjunction of *literals*. A literal is a variable or its negation. For the 3-SAT problem, all clauses contains at most three literals. We use 3-SAT as benchmark, because it is a well studied problem in the computer science. It is known to be $\mathcal{NP}$*-hard* and thus a computational challenge. Furthermore, it is based on boolean data types and can be solved with small memory space.

Our algorithm is a BSP implementation of a divide and conquer algorithm ([HO01], pp. 169–173). In each step a variable of the shortest clause is assigned and the remaining formula is checked recursively. If the assignment does not fulfill the clause, one of the remaining literals have to evaluate to true. Thus, not all possible combinations of the

$$(\bar{x}_7 \vee x_{17} \vee x_{28}) \wedge (\bar{x}_{15} \vee x_2 \vee x_{17}) \wedge (\bar{x}_{35} \vee \bar{x}_{10} \vee \bar{x}_{29}) \wedge (\bar{x}_{49} \vee x_{19} \vee x_{31}) \wedge (\bar{x}_5 \vee x_8 \vee x_{34})$$
$$\wedge (\bar{x}_{37} \vee x_{41} \vee x_{28}) \wedge (\bar{x}_{14} \vee x_{42} \vee x_9) \wedge (\bar{x}_{47} \vee \bar{x}_{46} \vee x_0) \wedge (\bar{x}_{44} \vee \bar{x}_{15} \vee x_{32}) \wedge (x_{37} \vee x_{41} \vee x_{21})$$
$$\wedge (\bar{x}_{48} \vee \bar{x}_{33} \vee x_{40}) \wedge (x_{11} \vee x_{16} \vee x_{34}) \wedge (\bar{x}_{48} \vee x_{21} \vee x_6) \wedge (\bar{x}_9 \vee \bar{x}_{10} \vee \bar{x}_{38}) \wedge (\bar{x}_{27} \vee \bar{x}_{35} \vee \bar{x}_{36})$$
$$\wedge (\bar{x}_{12} \vee \bar{x}_{45} \vee \bar{x}_{40}) \wedge (\bar{x}_3 \vee x_{20} \vee x_8) \wedge (x_{11} \vee x_{47} \vee x_9) \wedge (x_{23} \vee x_{22} \vee x_{14}) \wedge (\bar{x}_7 \vee \bar{x}_{40} \vee \bar{x}_{42})$$
$$\wedge (\bar{x}_{43} \vee \bar{x}_1 \vee \bar{x}_{34}) \wedge (\bar{x}_{26} \vee x_{17} \vee x_{30}) \wedge (x_{46} \vee x_{34} \vee x_5) \wedge (\bar{x}_{10} \vee \bar{x}_{26} \vee x_{40}) \wedge (\bar{x}_{32} \vee \bar{x}_{46} \vee \bar{x}_{44})$$
$$\wedge (\bar{x}_{15} \vee \bar{x}_{36} \vee x_{35}) \wedge (\bar{x}_{45} \vee x_{26} \vee x_1) \wedge (\bar{x}_{27} \vee x_{14} \vee x_9) \wedge (\bar{x}_{37} \vee \bar{x}_{38} \vee x_{45}) \wedge (\bar{x}_{32} \vee \bar{x}_3 \vee x_{23})$$
$$\wedge (\bar{x}_{11} \vee \bar{x}_{44} \vee x_{49}) \wedge (\bar{x}_{31} \vee \bar{x}_{20} \vee \bar{x}_{14}) \wedge (x_7 \vee x_{41} \vee x_{23}) \wedge (\bar{x}_{48} \vee x_{29} \vee x_3) \wedge (\bar{x}_8 \vee x_{44} \vee x_{27})$$
$$\wedge (\bar{x}_{32} \vee \bar{x}_{46} \vee \bar{x}_0) \wedge (\bar{x}_{15} \vee x_{26} \vee x_0) \wedge (\bar{x}_{10} \vee \bar{x}_{16} \vee \bar{x}_{34}) \wedge (\bar{x}_{41} \vee \bar{x}_{14} \vee x_{44}) \wedge (\bar{x}_{18} \vee \bar{x}_{26} \vee x_{29})$$
$$\wedge (x_2 \vee x_{27} \vee x_{11}) \wedge (\bar{x}_{10} \vee \bar{x}_{32} \vee x_{47}) \wedge (\bar{x}_5 \vee \bar{x}_8 \vee x_{36}) \wedge (\bar{x}_{36} \vee \bar{x}_6 \vee x_{12}) \wedge (\bar{x}_1 \vee x_{25} \vee x_{15})$$
$$\wedge (\bar{x}_{23} \vee \bar{x}_{37} \vee x_{45}) \wedge (\bar{x}_{12} \vee \bar{x}_{23} \vee \bar{x}_7) \wedge (\bar{x}_{35} \vee \bar{x}_{41} \vee \bar{x}_{20}) \wedge (\bar{x}_{36} \vee \bar{x}_{18} \vee x_2) \wedge (\bar{x}_{30} \vee \bar{x}_{49} \vee x_{34})$$
$$\wedge (\bar{x}_6 \vee \bar{x}_{25} \vee x_{28}) \wedge (\bar{x}_{41} \vee \bar{x}_{44} \vee x_{28}) \wedge (\bar{x}_5 \vee x_{24} \vee x_{32}) \wedge (\bar{x}_{44} \vee \bar{x}_4 \vee x_6) \wedge (\bar{x}_6 \vee \bar{x}_5 \vee x_{27})$$
$$\wedge (\bar{x}_{47} \vee \bar{x}_{10} \vee x_{30}) \wedge (\bar{x}_{36} \vee x_{15} \vee x_{31}) \wedge (\bar{x}_{23} \vee x_{47} \vee x_0) \wedge (\bar{x}_{45} \vee x_{17} \vee x_{22}) \wedge (\bar{x}_{29} \vee \bar{x}_{49} \vee x_{47})$$
$$\wedge (\bar{x}_{20} \vee \bar{x}_1 \vee x_{38}) \wedge (x_{23} \vee x_{46} \vee x_{41}) \wedge (\bar{x}_{35} \vee x_{29} \vee x_3) \wedge (\bar{x}_4 \vee \bar{x}_0 \vee x_{27}) \wedge (\bar{x}_{46} \vee \bar{x}_{41} \vee x_{31})$$
$$\wedge (\bar{x}_{21} \vee x_{36} \vee x_{15}) \wedge (\bar{x}_{42} \vee \bar{x}_{33} \vee x_{41}) \wedge (\bar{x}_{39} \vee \bar{x}_{19} \vee x_{38}) \wedge (\bar{x}_{48} \vee x_{28} \vee x_5) \wedge (\bar{x}_{40} \vee \bar{x}_2 \vee x_{38})$$
$$\wedge (\bar{x}_{15} \vee \bar{x}_{11} \vee x_{42}) \wedge (x_{23} \vee x_{21} \vee x_2) \wedge (\bar{x}_{44} \vee x_{46} \vee x_{42}) \wedge (\bar{x}_{36} \vee x_{44} \vee x_{45}) \wedge (\bar{x}_8 \vee x_{25} \vee x_4)$$
$$\wedge (\bar{x}_2 \vee \bar{x}_{12} \vee x_{22}) \wedge (\bar{x}_{33} \vee x_4 \vee x_{12}) \wedge (x_{11} \vee x_{38} \vee x_{12}) \wedge (x_{21} \vee x_{49} \vee x_{36}) \wedge (x_{18} \vee x_8 \vee x_{45})$$
$$\wedge (\bar{x}_{23} \vee \bar{x}_{26} \vee x_7) \wedge (\bar{x}_{27} \vee x_6 \vee x_{20}) \wedge (\bar{x}_{24} \vee x_7 \vee x_{49}) \wedge (x_{19} \vee x_{49} \vee x_3) \wedge (x_{26} \vee x_{35} \vee x_{12})$$
$$\wedge (\bar{x}_{24} \vee x_{30} \vee x_{25}) \wedge (\bar{x}_{43} \vee \bar{x}_{31} \vee x_{38}) \wedge (\bar{x}_{19} \vee \bar{x}_9 \vee x_{40}) \wedge (\bar{x}_{27} \vee x_{48} \vee x_{34}) \wedge (x_0 \vee x_{43} \vee x_{33})$$
$$\wedge (\bar{x}_{10} \vee x_{34} \vee x_{38}) \wedge (\bar{x}_{49} \vee \bar{x}_{41} \vee \bar{x}_6) \wedge (\bar{x}_{23} \vee x_6 \vee x_{46}) \wedge (\bar{x}_{12} \vee \bar{x}_{47} \vee x_4) \wedge (\bar{x}_8 \vee \bar{x}_{19} \vee \bar{x}_{22})$$
$$\wedge (\bar{x}_{18} \vee x_{16} \vee x_1) \wedge (x_{10} \vee x_{22} \vee x_{20}) \wedge (\bar{x}_{44} \vee x_{29} \vee x_{14}) \wedge (\bar{x}_{23} \vee x_{25} \vee x_{10}) \wedge (\bar{x}_{12} \vee x_{32} \vee x_{37})$$
$$\wedge (\bar{x}_{26} \vee \bar{x}_6 \vee x_{43}) \wedge (x_{40} \vee x_{48} \vee x_1) \wedge (\bar{x}_{17} \vee \bar{x}_{36} \vee x_{11}) \wedge (\bar{x}_1 \vee \bar{x}_{25} \vee x_{11}) \wedge (\bar{x}_{18} \vee x_6 \vee x_{31})$$
$$\wedge (\bar{x}_{21} \vee x_{10} \vee x_{32}) \wedge (\bar{x}_{19} \vee x_{11} \vee x_7) \wedge (\bar{x}_{47} \vee x_{39} \vee x_{15}) \wedge (\bar{x}_1 \vee \bar{x}_{23} \vee \bar{x}_{10}) \wedge (\bar{x}_{16} \vee x_{25} \vee x_{36})$$
$$\wedge (\bar{x}_{13} \vee \bar{x}_{18} \vee x_{45}) \wedge (x_4 \vee x_{46} \vee x_{35}) \wedge (\bar{x}_{28} \vee \bar{x}_8 \vee x_{18}) \wedge (x_{31} \vee x_3 \vee x_{27}) \wedge (\bar{x}_{33} \vee \bar{x}_{45} \vee x_{19})$$
$$\wedge (\bar{x}_3 \vee \bar{x}_{35} \vee \bar{x}_{12}) \wedge (\bar{x}_{14} \vee \bar{x}_{36} \vee x_{44}) \wedge (\bar{x}_{20} \vee x_{28} \vee x_{22}) \wedge (\bar{x}_5 \vee \bar{x}_{39} \vee x_6) \wedge (\bar{x}_{41} \vee \bar{x}_{28} \vee x_{30})$$
$$\wedge (\bar{x}_{35} \vee x_{23} \vee x_{30}) \wedge (\bar{x}_{44} \vee \bar{x}_{36} \vee \bar{x}_0) \wedge (\bar{x}_5 \vee \bar{x}_{28} \vee x_2) \wedge (\bar{x}_{27} \vee \bar{x}_{49} \vee x_{26}) \wedge (x_{43} \vee x_{25} \vee x_4)$$
$$\wedge (\bar{x}_{16} \vee \bar{x}_{47} \vee x_{48}) \wedge (\bar{x}_{39} \vee \bar{x}_6 \vee x_{11}) \wedge (\bar{x}_{11} \vee \bar{x}_{47} \vee x_{30}) \wedge (\bar{x}_{41} \vee x_{31} \vee x_{26}) \wedge (\bar{x}_{26} \vee \bar{x}_9 \vee x_0)$$
$$\wedge (\bar{x}_{48} \vee x_5 \vee x_9) \wedge (\bar{x}_{23} \vee x_7 \vee x_{42}) \wedge (x_{22} \vee x_{30} \vee x_0) \wedge (\bar{x}_{46} \vee x_{10} \vee x_{37}) \wedge (\bar{x}_{27} \vee \bar{x}_{12} \vee x_{25})$$
$$\wedge (\bar{x}_{39} \vee \bar{x}_{41} \vee x_{11}) \wedge (\bar{x}_2 \vee x_{38} \vee x_{45}) \wedge (x_{16} \vee x_{40} \vee x_{45}) \wedge (x_{22} \vee x_{20} \vee x_{12}) \wedge (\bar{x}_{13} \vee \bar{x}_0 \vee \bar{x}_{37})$$
$$\wedge (x_{19} \vee x_{17} \vee x_5) \wedge (\bar{x}_{49} \vee \bar{x}_8 \vee x_{19}) \wedge (\bar{x}_{31} \vee \bar{x}_{17} \vee x_9) \wedge (\bar{x}_{20} \vee \bar{x}_{33} \vee x_{48}) \wedge (\bar{x}_{34} \vee x_{22} \vee x_{43})$$
$$\wedge (\bar{x}_{18} \vee x_{39} \vee x_{33}) \wedge (\bar{x}_0 \vee \bar{x}_{11} \vee x_5) \wedge (\bar{x}_1 \vee \bar{x}_6 \vee x_5) \wedge (\bar{x}_{19} \vee x_{31} \vee x_{33}) \wedge (\bar{x}_{11} \vee \bar{x}_{28} \vee x_{42})$$
$$\wedge (\bar{x}_{48} \vee x_1 \vee x_{23}) \wedge (\bar{x}_3 \vee x_9 \vee x_{39}) \wedge (x_{10} \vee x_4 \vee x_{11}) \wedge (\bar{x}_2 \vee \bar{x}_{30} \vee x_{46}) \wedge (\bar{x}_{22} \vee x_{42} \vee x_{20})$$
$$\wedge (\bar{x}_{40} \vee \bar{x}_{35} \vee \bar{x}_{49}) \wedge (\bar{x}_7 \vee \bar{x}_{41} \vee \bar{x}_{23}) \wedge (x_{38} \vee x_{44} \vee x_6) \wedge (\bar{x}_{44} \vee x_{36} \vee x_6) \wedge (x_{40} \vee x_{39} \vee x_7)$$
$$\wedge (\bar{x}_{49} \vee \bar{x}_9 \vee \bar{x}_7) \wedge (\bar{x}_4 \vee \bar{x}_{38} \vee \bar{x}_{13}) \wedge (\bar{x}_{21} \vee \bar{x}_{23} \vee \bar{x}_{42}) \wedge (\bar{x}_{35} \vee x_{39} \vee x_{34}) \wedge (x_{16} \vee x_{48} \vee x_{40})$$
$$\wedge (\bar{x}_{31} \vee x_6 \vee x_{23}) \wedge (\bar{x}_{29} \vee \bar{x}_7 \vee \bar{x}_8) \wedge (\bar{x}_{40} \vee \bar{x}_{12} \vee \bar{x}_9) \wedge (\bar{x}_{32} \vee x_{25} \vee x_{30}) \wedge (\bar{x}_{21} \vee \bar{x}_{38} \vee x_{16})$$
$$\wedge (\bar{x}_{20} \vee x_{27} \vee x_2) \wedge (\bar{x}_{13} \vee x_{45} \vee x_{22}) \wedge (x_{28} \vee x_{15} \vee x_{18}) \wedge (\bar{x}_{31} \vee \bar{x}_{43} \vee x_{41}) \wedge (\bar{x}_{23} \vee x_9 \vee x_{22})$$
$$\wedge (\bar{x}_0 \vee \bar{x}_{31} \vee \bar{x}_{20}) \wedge (\bar{x}_7 \vee \bar{x}_{43} \vee \bar{x}_{38}) \wedge (x_{38} \vee x_{10} \vee x_8) \wedge (\bar{x}_{45} \vee x_{13} \vee x_{18}) \wedge (\bar{x}_{41} \vee x_{43} \vee x_{45})$$
$$\wedge (\bar{x}_{28} \vee x_{22} \vee x_{36}) \wedge (x_{31} \vee x_{24} \vee x_{19}) \wedge (\bar{x}_{42} \vee \bar{x}_{11} \vee x_{13}) \wedge (\bar{x}_{35} \vee \bar{x}_{17} \vee x_{45}) \wedge (\bar{x}_{25} \vee \bar{x}_9 \vee x_{13})$$
$$\wedge (\bar{x}_1 \vee \bar{x}_{29} \vee x_4) \wedge (\bar{x}_{17} \vee x_5 \vee x_{45}) \wedge (\bar{x}_{25} \vee \bar{x}_{43} \vee x_1) \wedge (\bar{x}_7 \vee \bar{x}_{10} \vee x_{19}) \wedge (\bar{x}_{30} \vee x_2 \vee x_{15})$$
$$\wedge (\bar{x}_{21} \vee \bar{x}_8 \vee x_{38}) \wedge (\bar{x}_{48} \vee \bar{x}_{41} \vee x_{43}) \wedge (\bar{x}_{44} \vee \bar{x}_{43} \vee x_{30}) \wedge (\bar{x}_{30} \vee \bar{x}_{10} \vee x_{49}) \wedge (\bar{x}_{31} \vee \bar{x}_{45} \vee x_1)$$
$$\wedge (\bar{x}_5 \vee \bar{x}_6 \vee x_{16}) \wedge (\bar{x}_{31} \vee x_{18} \vee x_{47}) \wedge (\bar{x}_9 \vee x_{19} \vee x_{38}) \wedge (\bar{x}_{21} \vee \bar{x}_{36} \vee x_{37}) \wedge (\bar{x}_{30} \vee \bar{x}_{47} \vee x_8)$$
$$\wedge (x_{39} \vee x_{11} \vee x_6) \wedge (\bar{x}_{23} \vee \bar{x}_3 \vee x_8) \wedge (\bar{x}_{21} \vee x_{48} \vee x_{32}) \wedge (\bar{x}_{11} \vee x_{42} \vee x_9) \wedge (\bar{x}_{29} \vee \bar{x}_9 \vee x_{24})$$
$$\wedge (x_{45} \vee x_{46} \vee x_{30}) \wedge (\bar{x}_6 \vee x_{26} \vee x_{12}) \wedge (\bar{x}_{44} \vee \bar{x}_{34} \vee x_{31}) \wedge (\bar{x}_{49} \vee x_{33} \vee x_8) \wedge (x_1 \vee x_{33} \vee x_{29})$$
$$\wedge (x_2 \vee x_{15} \vee x_1) \wedge (\bar{x}_{17} \vee \bar{x}_{11} \vee x_{44}) \wedge (x_{32} \vee x_{36} \vee x_9) \wedge (\bar{x}_{17} \vee x_6 \vee x_{42}) \wedge (\bar{x}_{21} \vee \bar{x}_{18} \vee x_{43})$$
$$\wedge (\bar{x}_{30} \vee \bar{x}_{26} \vee \bar{x}_{41}) \wedge (\bar{x}_2 \vee \bar{x}_{39} \vee x_7) \wedge (\bar{x}_{22} \vee \bar{x}_{30} \vee x_{37})$$

Figure 5.12: 3-SAT Instance uuf050-218, 50 Variables, 218 Clauses, Unsatisfiable

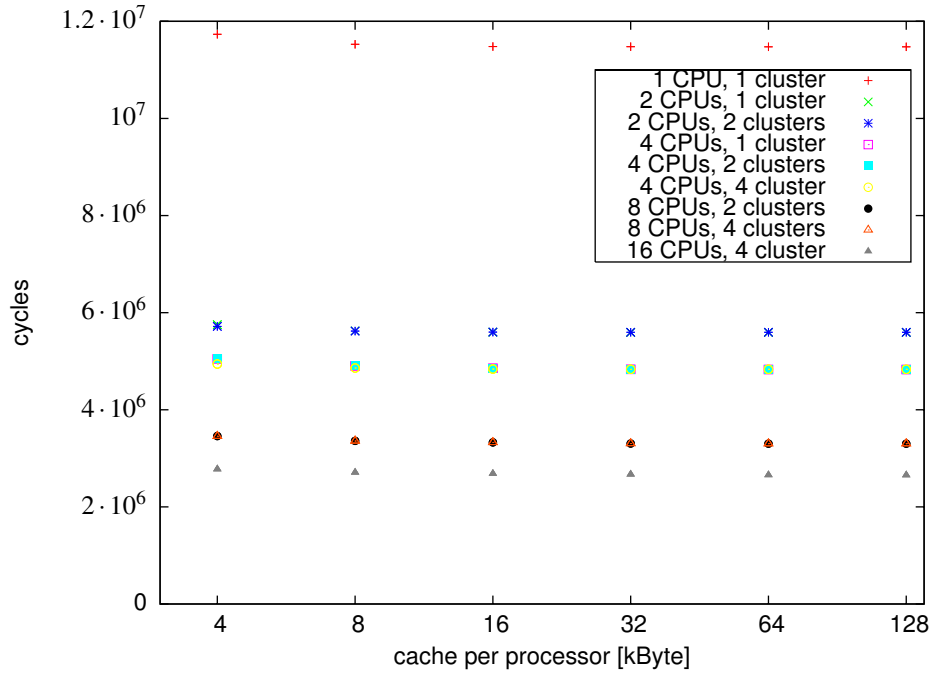


Figure 5.13: 3-Satisfiability: Processor Cache

variable assignment have to be checked.

In the first step processor 0 executes the algorithm sequentially until $c \cdot p$ subproblems are created for a constant c (we use $c = 4$ in our experiments). These branches are distributed evenly among the processors. Then all processors try to find a solution sequentially for their subproblems. If a solution is found, it is broadcasted and the computation stops at the end of the actual superstep.

Our objective was not to develop the fastest 3-SAT solver, or concentrate on load balancing, we just want to compare our architecture with standard technologies. Furthermore, we want to evaluate the influence of system parameter, for instance, cache sizes, for the running time.

For the benchmark we use 3-SAT instances from the SATLIB-benchmark. The Satisfiability Library SATLIB [HS00] is a collection of benchmark problems, solvers, and tools for SAT related research. Instances can be downloaded from the website of the benchmark [SAT00]. In the following, we use file uuf050-218.cnf of the archive uuf50-218.tar.gz. The instance (Figure 5.12) was generated uniformly at random and proven to be unsatisfiable. It was converted from a format called DIMACS into our own data structure, and linked directly into the program, as our simulators do not support input yet.

First, we evaluate the influence of the processor cache for different numbers of processors and topology (Figure 5.13). For a single processor, more than the default 16 kb cache

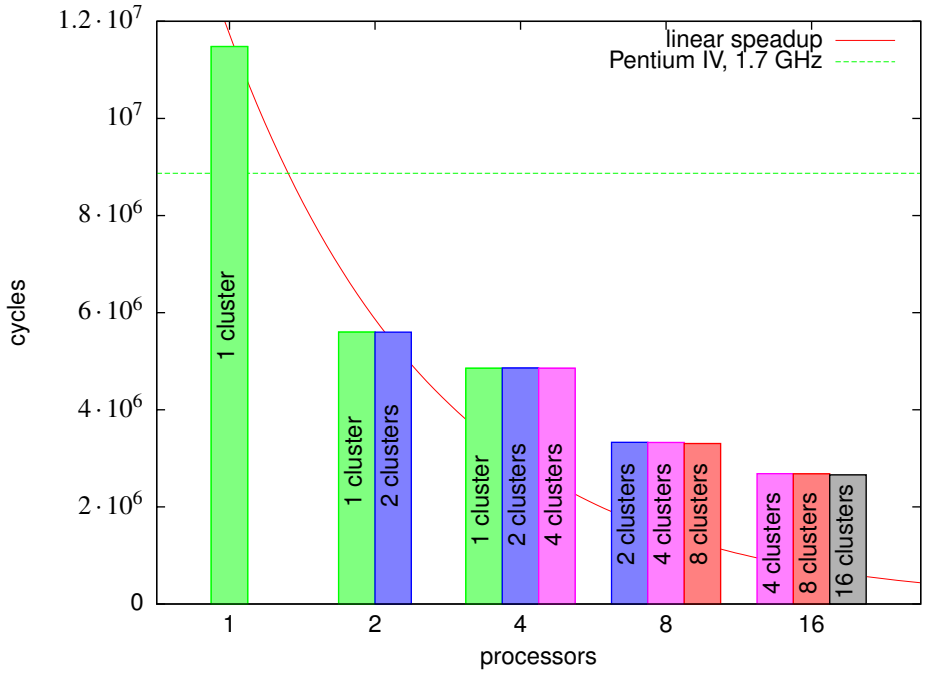


Figure 5.14: 3-Satisfiability: Number of Processor

is not needed, the program runs only 0.053 % faster when increasing the cache to 128 kb. The efficiency of the cache is the reason. We have 7,106,264 hits and only 1,367 misses for reading (99.9808 % hit rate), and 222,787 hits and 1,908 misses for writing (99.1508 % hit rate). Using only 8 kb (4 kb) cache, the number of cycles increases by 0.41 % (2.2 %) due to the lower hit rate (99.5528 % read, 92.675 % write for a 4 kb cache).

If there are four processors sharing the bus, the cache becomes more important, because the memory bandwidth per processor decreases. However, the 3-Sat benchmark has a small working set. The performance decreases by only 1 % (4 %) if the cache size is decreased from 16 kb to 8 kb respectively 4 kb.

Next, we look at different system configurations of the processors. Our algorithm does not scale well, because the initial work distribution is done by a single processor. If we increase the number of processors, this part will dominate the total execution time (Figure 5.14). Scalability was not our intention for this test. We want to analyse the influence of the topology, i. e., the differences between using shared memory and using message passing communication in different switch boxes. As there is almost no communication in this benchmark, we expected that it is faster to have more clusters, thus less processors competing for the shared memory bus.

But there is almost no difference in the running time. The shared memory is not the bottleneck in this application, because the cache is very efficient and most memory

accesses can be executed by the cache itself.

Sorting. The second application benchmark is sorting of integer numbers. We use an implementation of SAMPLESORT, a parallel sorting algorithm similar to the sequential QUICKSORT. QUICKSORT partitions the data into two parts, smaller and bigger numbers, using a single splitting element. Then these parts are sorted recursively. The efficiency of the algorithm is dependent on a good splitting element which creates two partitions of approximately the same size.

SAMPLESORT for p processors uses $p-1$ splitting elements and partitions the data into p sets. Each processor is responsible for one set, thus each processor sends all keys to the appropriate destination. Next, all processor sort their data locally. The efficiency is dependent on the splitting elements, a good choice leads to a good load balancing. To get the splitting elements, each processor chooses a number of random samples of the data and sends these samples to processor 0. Processor 0 sorts them locally and broadcast the perfect splitting elements for the samples (cf. Section 3.1.3).

In our benchmark we sort random 32 bit integer numbers. Each processor sends 64 samples to processor 0, and all data send to the destination processor is combined in messages of size 1024 bytes each. These parameters are not optimal for the problem – in fact we do not need samples at all as we sort numbers chosen uniformly at random – but our intention is to evaluate the architecture and not to find the best sorting algorithm.

For the architecture we use all valid combinations of the following parameters: width and height 1, 2, 3, or 4 clusters, 1, 2, or 4 of processors per cluster, 0 kb, 4 kb, 8 kb, 16 kb, 32 kb, 64 kb, or 128 kb of cache per processor, 16 bytes, 32 bytes, or 64 bytes cache line size, with amba matrix or standard bus system.

The results of these 2,000 simulations (running several hours on up to 35 fast workstations in total) is shown in Figure 5.15. Each system size, i. e., each total number of processors, is shown in another color. The different configurations (e. g., topology, cache sizes) lead to small differences in the execution time, but the total number of processors has much more impact on the execution speed. For small data size the overhead of the sample sort algorithm compared to sequential QUICKSORT is the crucial factor. But for data sizes larger than 2 kByte, the parallel SAMPLESORT outperforms QUICKSORT. The larger the data set is, the more processors can be used efficiently. The results do not show any unexpected effects, for example, network congestion or other problems.

We ran the same benchmark on a standard processor (Pentium IV, 1.7 GHz) to classify the performance of the system. The parallel system on a chip is much faster (counting cycles) even using only a single CPU, because it contains on chip memory, whereas the Pentium has to load and store all data through the memory hierarchy using slow off chip memory.

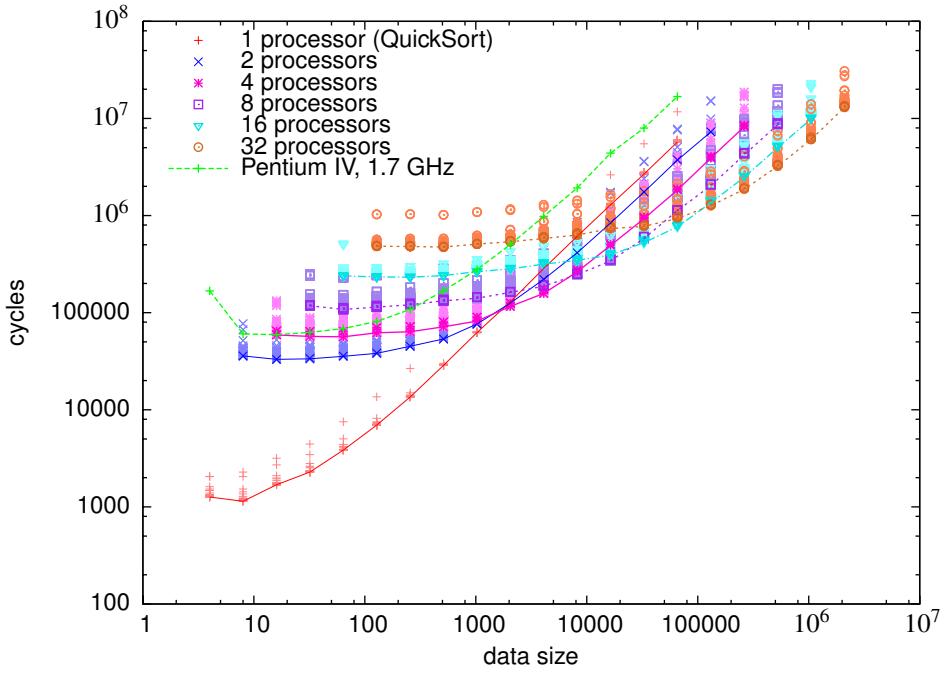


Figure 5.15: Sorting: Samplesort

5.6 Conclusions

In this chapter we proposed a parallel system on a single chip suiteable for BSP computing. Although our design is just a prototype for evaluation the potential of parallel on chip systems, it outperforms classical CPUs in some benchmarks. Replacing the simple processors cores by more powerfull ones will significantly increase the performance of the total system.

We show two things: first, the BSP model can assist hardware developers in designing parallel systems for general use. The fast synchronization instruction of the processing elements, and the communication protocol of the interconnection network are just two examples of optimizations for the requirements of an efficient BSP implementation.

Furthermore, we show that the BSP model is applicable for this kind of parallel system, which differs from the classical parallel computers the model was designed for.

Chapter 6

BSP for Heterogeneous Workstation Clusters

The Bulk Synchronous Parallel Model (BSP) was designed for classical monolithic parallel machines. In the last chapter we showed that the model is also applicable for parallel systems on a chip. In this chapter we look at the other end of the spectrum of parallel systems, classical workstations interconnected by a local area network. We show, how BSP can be implemented efficiently on such architectures, even if the computers spend different computation power, which even might change during the execution.

We use virtual processors, and balance the load online by migrating these processors to less loaded computers if the load changes. We have developed a technique to migrate processes during execution. This is used for load balancing, but have many other applications besides BSP.

In the following, we describe our view of heterogenous workstation cluster. Then we show the techniques of migration a virtual processor to another computer without using special adaption of the operating system. Next we present and evaluate load balancing strategies to show the benefit of the system. Parts of this work has been published in [Bon07].

6.1 Heterogeneous Workstation Cluster

A workstation cluster consists of computers, in the majority of cases personal computers (PCs). These computers are interconnected by an arbitrary network, which supports the TCP/IP protocol. In most cases a (Fast-/Gigabit-) Ethernet network is used, but any other physical layer is possible, e. g., InfiniBand. For demonstration purpose, we restrict ourselves to PCs with processors using the 32 bit x86-32 instruction set (e. g., Intel Pentium, AMD Athlon).

The workstations are heterogeneous. This means they are different computers (e. g.,

manufactures, attached devices), providing different computational power. The usable power is changing dynamically due to other users, some computer may even break down. However, they are not complete heterogeneous, they have the same architecture (e. g., Intel 80x86), the same operating system (e. g., Linux), and sometimes a global filesystem (NFS) and a centered administration.

Comparison with Monolithic Parallel Computers

Typically users of classical parallel systems have exclusive access to parts of resources, i. e., they can allocate partitions of the system for a certain amount of time or there is a batch system which allows submitting jobs that are executed when the needed resources are available. There are several advantages of this approach for commercially used supercomputers: each job obtains all resources, e. g., can use the whole memory, less security problems, and the accounting is easier.

Besides the exclusive access all processors of such system are very similar, in most cases they are even the same. In the BSP cost model the length of a superstep is given by the maximum of the work over all processors, regardless of the work of all other processors. Thus to get an efficient BSP algorithm the work should be distributed evenly among all processor. If the computer have processors of different speed or, in the dynamic case of changing available computation power like in our scenario, the standard BSP model is not adequate. For the efficient execution of jobs on such dynamic system scheduling and load balancing have to be considered.

Our Approach

The usage of virtual processors is one way to cope with different processor's speeds: the BSP program is executed on virtual processors, which are simulated by the real computers. The faster a computer is, the more virtual processors are assigned to it. For static processor speeds this can be done by a simple scheduler, but if the available computation power is changing, or the work is not distributed evenly among the virtual processors of the BSP program, a more sophisticated online load balancing system is needed.

6.2 Related Work

There exist several approaches to use the idle time of workstation in offices. First, there are programs for solving specialized problems, SETI@home (search for extraterrestrial intelligence [set08]) and distributed.net (e. g., breaking cryptographic keys [dis08]) are two of the most famous ones. These systems scan a large data space by dividing it into small pieces of data, sending them as jobs to the clients, and waiting for the results. There is no communication during the calculation, or the communication is combined with the

job data and sent through the central server as in the BSP implementation by Bayanihan [Sar99]. These loosely coupled systems can efficiently be executed on PCs connected by the Internet.

Achieving fault tolerance is straight forward: If a client does not return the result, a timeout occurs and the job is assigned to another client. Although only a small, fixed number of problems can be solved, these systems are very powerful in terms of computation power because of the huge number of clients taking part.

Second, there are job scheduling and migration systems. Here you can start jobs, either directly or using a batch system. These jobs are executed on idle nodes and may migrate to other nodes if the load in the system changes. Condor [LTBL97] is one example for such a system implemented in user space, MOSIX [BL98] is a module for the Linux kernel which allows migrations of single processes. Kerrighed [MGLV04] goes beyond this and provides a single system image of a workstation cluster. It supports migration of processes and also communication via shared memory.

Checkpointing (of sequential programs) is used either for fault tolerance or to speed up the startup of applications like the editor Emacs [Sta81] or the LaTeX-system do.

Emacs implements a function `unexec`, which does the opposite of the normal `exec` function[1] and stores the actual state of a process together with the program code into one executable. If this executable is started later, the process is restored and continues.

Cao, Li, and Guo have implemented process migration for MPI applications based on coordinated checkpoints [CLG05]. They use an MPI checkpointing system for the LAM MPI implementation [SSB+05] based on Berkeley Lab's Linux Checkpoint and Restart [Due05], a kernel level checkpointing system. As MPI stores the location of the processes, they have to modify the checkpoints before restarting.

An implementation for migration of single threads, even in a heterogeneous network with different CPU types, is proposed by Dimitrov and Rego in [DR98]. They extend the C language and implement a preprocessor which automatically inserts code for keeping track of all local data. Additionally, the state in each function, i. e., the actual execution point, is stored and a switch statement, which branches to the right position depending on the state if a thread is recreated, is inserted at the beginning of each function. These changes lead to an overhead of 61 % in their example, mainly due to the indirect addressing of local variables. A similar approach for Java is JavaGo [SSY00], used in the BSP web computing library PUBWCL [BGM06].

Most of these systems support only independent jobs (Condor can schedule parallel jobs using MPI or PVM, but these jobs do not take part in load balancing and migration) or affect parts of the operation system kernel like Kerrighed, thus the user do not only need administrative access to the machines, they furthermore have to install a Linux kernel with special patches, which has some drawbacks (usable kernel versions are limited, e. g., not up-to-date).

[1] The `exec` function replaces the current process image with a new process image read from a file.

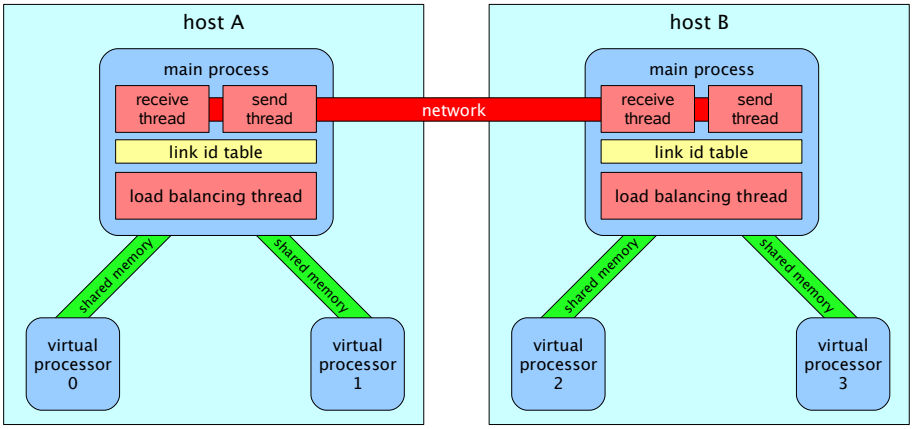


Figure 6.1: Structure of PUB with Migratable Processes

There exists also some work about checkpointing threaded processes [DL01], which is useful to migrate parallel programs for symmetric multiprocessor (SMP) machines.

Hill, Donaldson, and Lanfear [HDL98] describe a system that supports migrations for parallel programs written with the BSPlib library. They use the unexec function to store checkpoints to disks. If the load should be balanced, all processors write a checkpoint to disk, and all processes are terminated. Next they are restarted on the p least loaded computers. Such a migration requires a lot of disk accesses, thus it took a long time. However, after a balancing, the schedule is optimal, whereas our implementation migrates single jobs, which leads to a small improvement of the schedule, but at much lower costs.

6.3 Migration of Virtual Processors

Using PUB virtual processors can be created by the function `bsp_migrate_dup` (cf. the user guide and reference of PUB [BGOW07]). This will start one new process for each virtual processor. Figure 6.1 gives an overview of the processes, their threads and the connections. The main process consists of three additional threads: The *receive thread* reads all messages from the network and sends them to the right destination by either calling the proper message-handler or putting the message into a shared memory queue to a virtual processor. The *send thread* sends all messages from a shared memory send queue to the network. The *load balancing thread* gathers information about the load of the system and decides when to migrate a virtual processor. We have implemented several algorithms for load balancing (cf. Section 6.4). The virtual processors are connected to the main process by two shared memory message queues, one for each direction.

The link-id-table is the routing table. For each virtual processor of the system it stores the actual execution host and the link id (used to distinguish the processes). This table

does not have to be up-to-date on all hosts, for example if a process was migrated, but then the destination host stored in the table knows the new location and will forward the message to it. Thus a message sent to the location stored in the table will reach its destination, although it might take some additional hops. The maximal number of these hops is unlimited, if a process does not migrate at the moment, the message may be forwarded by all hosts in the worst case. If the process is migrating, it may escape the message, such that the message is always following the process. However, this scenario is very unlikely.

6.3.1 Procedure of the Migration of a Virtual Processor

When the load balancer decides to migrate one process to another host, the following steps are executed:

1. The load balancer writes the destination host id to a migration queue.

2. The first process which enters the synchronization dequeues this id. It sends a message to the main process.

3. The receive thread reads this message, marks the virtual processor as *gone* and sends an acknowledgment to it. From now on it will not send any messages to the process. All messages for this destination will be stored in a buffer in the main process.

4. The process to migrate reads this acknowledgment, opens a TCP/IP server socket and sends a message to the destination host.

5. The destination host creates two shared memory queues and a new child process.

6. This new process connects directly via TCP/IP to the old virtual processor.

7. The context of the process is transfered from the old host to the new destination. The details of this step are described in Section 6.3.2. After this the old process terminates.

8. The new process sends its new link id to the main process on the original host.

9. This host updates its link table and sends all messages arrived during the migration to the new destination. If any further messages arrive for this virtual processor, they will be forwarded to the new location. Optionally the migration can be broadcasted to reduce the number of forwarded messages.

6.3.2 Migration of Linux Processes

This section deals with the technical aspects of migrating a Linux process from one computer to another. We consider the parts of a process and look at its interaction with the operating system kernel. The *configuration* of a process consists of a user space and a kernel space part. As we do not extend the kernel, we cannot access the data in the kernel space unless we use standard system functions; in particular we are not allowed to access the kernel stack.

In the following we have a closer look to the migratable parts in the user space, and the dependencies to the kernel, namely system calls and thread local storage.

Linux System Calls. The access to the operating system is done by *syscalls*. A syscall is like a normal function call, but additionally the following steps are performed: the CPU switches to *supervisor mode* (also known as *kernel mode* or *ring 0*) and sets the stack pointer to the kernel stack. This allows the kernel to use all instructions and deal with stack overflows of the user stack. There are two different ways for a system call in Linux: Traditionally, Linux uses a software interrupt, i. e., the number of the system function and its arguments are stored in processors registers, and an interrupt is generated by the assembler instruction `int 0x80`. This interrupt call is very slow on Intel Pentium IV processors, the time for a system call is much higher on a 2 GHz Pentium IV than on a 850 MHz Pentium III [Hay02] (see [SB05] for a detailed view on measuring Linux kernel execution times). Intel has introduced a more efficient mechanism for privilege switches, the `sysenter` instruction, first available in Pentium Pro processors (November, 1995). There was a hardware bug in early CPUs, so it took a long time until operating systems started using sysenter. Linux supports sysenter since version 2.5.53 from December, 2002.

To provide a portable interface to different CPUs, Linux uses a technique called *vsyscall*. The kernel maps a page of memory (4096 bytes) into the address space of user programs. This page contains the best implementation for system calls, thus programs just do a simple subroutine call to this page. The page is called *VDSO* (virtual dynamic shared object), its address can be found in the `maps` file in the proc-filesystem (cf. Table 6.1). The address is also noted in the ELF header of dynamically linked programs and can be viewed with the command `ldd`, here the page looks like a dynamic library called linux-gate.so.

Before Linux version 2.6.18 (September, 2006) the address of the page was fixed to 0xffffe000. This is the last usable page in the 32 bit address space; the page at 0xfffff000 is not used because illegal memory accesses (pointer underruns) should lead to a page fault. Recent Linux versions use a random address [KeN06] to complicate some kind of buffer overflow attacks. The C library stores the address in a variable, `_dl_sysinfo`. The kernel also stores the address because sysenter does not save a return address. Instead, the `sysreturn` instruction used to return to the userspace needs this address as an argument. This means in particular that no user process can change the address of the VDSO page,

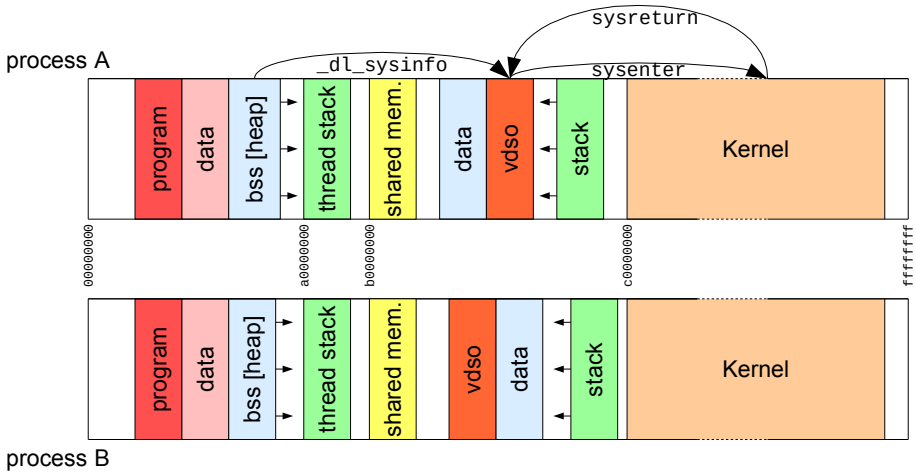


Figure 6.2: Virtual Address Space

because the kernel will always jump back to the old address.

If we migrate a process, we cannot guarantee that the source and the destination uses the same address, but we have to keep the old VDSO page. So we do not migrate this page and adapt the address in the C library, such that it uses the old page. If this is not possible because of user data is mapped to the same address (cf. Figure 6.2), we free the VDSO page, and change the `_dl_sysinfo` address to our implementation for system calls using the old interrupt method.

Randomization of the address of the VDSO page can be disabled with the sysctl command (`sysctl -w kernel.randomize_va_space=0`) or directly using the proc filesystem (`echo 0 > /proc/sys/kernel/randomize_va_space`).

Migratable Resources. When we migrate a process, we transfer all its resources to the new destination, namely the state of the processor, the used memory, and some other data stored inside the kernel, e. g., open files.

Processor state: The processor state contains all data stored in the processor, e. g., the registers. This data has to be migrated. We save the registers in the data segment[2] of our process. In [LTBL97], Litzkow and others suggest to use a signal handler for the migration. Inside a signal handler one can use all registers freely, i. e., the operating system has to store and restore all registers by itself and the code can become more portable. Older version of our implementation uses this technique, too. However, this is not compatible with the handling of signals in actual Linux kernels. If a signal is sent to a process, the kernel saves some data on the kernel stack, and writes a syscall to the kernel function

[2] we use the word “segment” for parts of memory, in the context of Intel processors segments are something different

Table 6.1: Example of a memory map read from maps of the proc-filesystem

virtual address	perm	offset			mapped file
08048000-08108000	r-xp	00000000	03:05	4432424	/tmp/migtest
08108000-08109000	rw-p	000bf000	03:05	4432424	/tmp/migtest
08109000-081c9000	rw-p	08109000	00:00	0	[heap]
a0000000-a0100000	rwxp	a0000000	00:00	0	
b0000000-b2000000	rw-s	00000000	00:07	305790982	/SYSV00000000 (deleted)
b7e00000-b7e21000	rw-p	b7e00000	00:00	0	
b7e21000-b7f00000	--p	b7e21000	00:00	0	
b7ffc000-b7ffd000	r-xp	b7ffc000	00:00	0	[vdso]
bffb7000-bffcd000	rw-p	bffb7000	00:00	0	[stack]

`sys_sigret` onto the user stack. So, when the signal handler returns, the kernel function restores the data from the kernel stack. We cannot modify or migrate this data on the kernel stack. See [Bar00] for more details.

Memory: The virtual address space of a Linux process contains different parts which must be handled separately. Table 6.1 shows the memory mapping of one virtual processor. This mapping can be read from the file `maps` in the Linux proc filesystem. First there is the program code segment, which is marked as read-only and is the execution file mapped into the address space. We do not need to migrate this because the program is the same on all nodes. Second, there are two data segments: One is for the initialized data; it is mapped from the execution file, but only copied on first write access. Next, there is the real data segment; this data segment grows when the application needs more memory and calls `malloc`. The size of this segment can be adjusted directly by `brk`. If the user allocates large blocks of data, the system creates new segments. We have to migrate the data segments and all these new segments.

The last segment type is the stack. It used to start at address `PAGE_OFFSET` defined in `page.h` of the Linux kernel, normally 0xc0000000[3], and grows downwards. For security reasons, Linux uses address space randomization [LWN05] since kernel version 2.6.12, i. e., the stack starts at a random address in the 64 KB area below `PAGE_OFFSET`, so we need to find the stack and restore it at the same address on the destination node. During the migration we use a temporary stack inside the shared memory.

Data stored in the Linux kernel: There is some data stored in the kernel, examples are information about open files and used semaphores. We have added support for regular files which exists on all the hosts, for instance, on a shared file system like the network file system (NFS). We gather information about all open files and try to reopen them on the destination host.

All other information stored in the kernel will be lost during migration, so the not use of semaphores, network connections, and other non migratable resources during the

[3]On 32 bit systems Linux splits the 4 GB address space into 3 GB user space from 0x00000000–0xbfffffff and 1 GB reserved for the kernel

migration (i. e., during the synchronization) is not allowed. If such resources are needed over more than one superstep, the migration of that virtual processor can temporarily be disabled.

Thread local storage: Although virtual processors are not allowed to use threads, we have to deal with *thread local storage* (TLS), because the ancestor of our processes, the main process, uses threads, i. e., all the processes in PUB are forked of a threaded process. TLS is used for global variables (accessible from all functions) that have different values for each thread. The C library uses this feature among other things for the `errno` variable: `errno` is a variable that is set to the error code of operations. Instead of using the function return value, the POSIX IEEE Std 1003.1 [IEE96] uses a global variable.

As all threads share the same address space, these variables must have different addresses for different threads, so the linker cannot insert a normal memory reference here. The Linux implementation uses a Thread Control Block (TCB), which stores all the information about TLS data. A pointer for the actual thread is accessible at the address 0 of the segment stored in the GS segment register (see [Dre05] for details). This segment is mapped to a part of the stack of the actual running thread. We create new processes for the migration by forking the receive thread, i. e., although we do not use threads inside virtual processors and have our own stack, we cannot free the stack of the receive thread, and furthermore we have to guarantee that the stack is at the same address on all hosts. Thus, we allocate our own stack at a fixed address and assign it to the receive thread with the function `pthread_attr_setstack` from the POSIX thread library.

6.3.3 Performance of the Migration

To measure the performance of migrations, we disable load balancing and force one process to migrate. Due to different local clocks, we send the process via all hosts in the network and back to the original location, i. e., a ping-pong test for two nodes and a round trip for more hosts.

We use two different clusters: four PCs with Intel Pentium D 3 GHz CPUs connected by a Gigabit Ethernet switch (D-Link DGS-1005D), and four PCs with Intel Core2Duo 2.6 GHz with a Cisco Switch respectively. The benchmark program allocates a block of memory and migrates one virtual processor from host 0 to $1, 2, \ldots p-1$ and back to host 0. These migrations are repeated 20 times. Figure 6.3 shows the mean time for one migration. The bandwidth grows up to 111 MB/s for large data sets, which is nearly optimal for Gigabit Ethernet. Without network, we achieve 850 MB/s (Core2Duo, 412 MB/s Pentium4 D) for migrations whereas a plain memory copy obtains 3.123 GB/s (respectively 1.926 GB/s). Considering that we always use TCP/IP connections with all the overhead of buffering and calls to the operating system, this is a very good performance. It is not worthwhile to optimize the case of local migrations, for instance, by using shared memory, because migrations to the same host are usefull only for this benchmark.

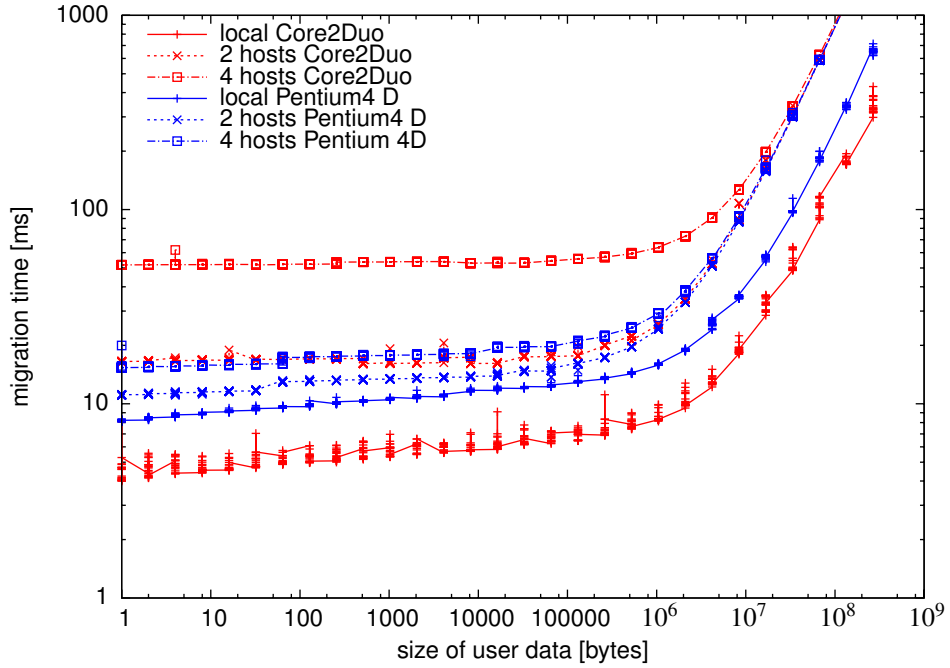


Figure 6.3: Time needed for Migrations

6.4 Load Balancing

In the last section, we have described a technique to migrate a virtual processor from one host to another. In this section, we will use this to balance the load of a workstation cluster. Before we describe the load balancing strategies, we will examine the typical load on workstation clusters and show two models for the load.

6.4.1 Load Measurements

To figure out the typical utilization of workstation, we measure the load and processor utilization of several workstation over weeks. On the one hand, we use this data for testing the load models, on the other hand, we extract load profiles and use them to simulate the load for our benchmarks.

First we have to define the term load of a computer. For Unix systems, the operation system often provides a load value for a computer. Normally, this is said to be the average number of processes with state ready-, running, or waiting for input/output.

Linux provides three load values, taking the average other 1 minute, 5 minutes, and 15 minutes. More precisely, the value is calculated by the following iterative formula every 5 seconds, given m the reporting period in minutes (1, 5, or 15), $n(t)$ the number of non

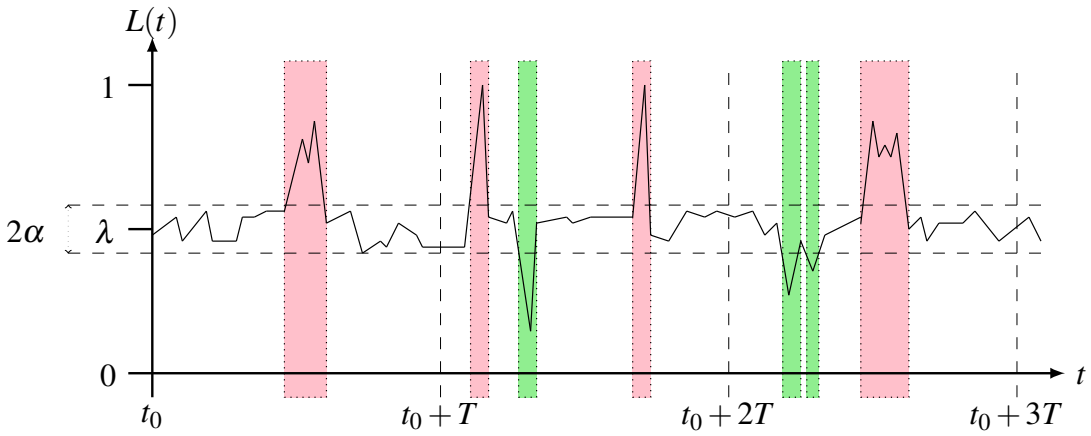


Figure 6.4: Example for the Constant Usage Interval Model

idling processes at time t, load$(t-1)$ the load in the period before t [Gun04]:

$$\text{load}(t) := \text{load}(t-1)e^{\frac{-5}{60m}} + n(t) \cdot (1 - e^{\frac{-5}{60m}})$$

This way to tame highly variable data is called *exponential smoothing*.

There are other measurements for the utilization of computers. Thomas Kunz analyzed the influence of different workload descriptions for load balancing [Kun91]. He examines the following indicators: the actual number of tasks in the running queue, rate of system calls, CPU context switch rate, percentage of idle CPU-time, size of free memory, and one minute load average or a combination of them.

6.4.2 Models for the Load

There are different ways to model the load of workstations. The first model is motivated by the observation that the CPU usage is typically nearly constant for larger time intervals.

Constant Usage Interval Model. The CPU usage typically shows a continuous pattern for quite a time, then changes abruptly, then again shows a continuous pattern for some time, and so on. The reason therefore is that many users often perform uniform activities (e. g., word processing, programming) or no activities at all (e. g., at night or during lunch break).

A given CPU usage graph (e. g., of the length of a week) can thus be split into blocks, in which the CPU usage is somewhat steady or shows a continuous pattern. These blocks typically have a duration of some hours, but also durations from only half an hour (e. g., lunch break) up to several days (e. g., a weekend) do occur.

Based on the above observations we have designed a model [BGM05] to describe and classify the external work load. We describe the CPU usage in such a block by a rather tight interval (with radius $\alpha \in \mathbb{R}$, $\alpha < \frac{1}{2}$) around a median load value ($\lambda \in \mathbb{R}$ with $0 \leq \lambda - \alpha$ and $\lambda + \alpha \leq 1$) and rates for the upper and lower deviation ($\beta^+ \in \mathbb{R}$, $\beta^+ < \frac{1}{2}$ resp. $\beta^- \in \mathbb{R}$, $\beta^- < \frac{1}{2}$) as illustrated in Figure 6.4. We will refer to such a block as a $(\lambda, \alpha, \beta^+, \beta^-, T)$-*load sequence* in the following.

In order to describe the frequency and duration of the deviations, we subdivide the load sequences into small sections of length T, called *load periods*. The values β^+ and β^- must be chosen such that the deviation rates never exceed them for an arbitrary starting point of a load period within the load sequence.

If a superstep is executed completely within a $(\lambda, \alpha, \beta^+, \beta^-, T)$-load sequence, the factor between the minimal and maximal possible duration is at most $q' \in \mathbb{R}^+$ with:

$$q' := \frac{1 - (1 - \beta^-)(\lambda - \alpha)}{1 - (\beta^+ + (1 - \beta^+)(\lambda + \alpha))}$$

For $q \in \mathbb{R}^+$, $q \geq q'$ we call a $(\lambda, \alpha, \beta^+, \beta^-, T)$-load sequence *q-bounded*.

This result guarantees that the running times of BSP processes, optimally scheduled based on the load of the previous superstep, differ at most by a factor q^2 within a load sequence. All our load balancing strategies monitor the past and use this information to predict the future.

Evaluating the Collected Data. When sectioning a given CPU usage sequence into load sequences, our goal is to obtain load sequences with a q-boundedness as small as possible and a duration as long as possible, while the rate of unusable time intervals should be as small as possible. Obviously, these three optimization targets depend on each other. We have processed the data collected from the monitored PCs (over 6.8 million samples) with a Perl program which yields an approximation for this non-trivial optimization problem.

The average idle time over a week ranged from approx. 35% up to 95%, so there is obviously a huge amount of unused computation power. Time intervals of less than half an hour and such where the CPU is nearly fully utilized by the user or its usage fluctuates too heavily, are no candidates for a load sequence. The rate of wasted idle time in such intervals is less than 3%.

Choosing suitable values for the parameters of the load sequences, it was possible to partition the given CPU usage sequences into load sequences such that the predominant part of the load sequences was 1.6-bounded. On most PCs, the average duration of a load sequence was 4 hours or even much longer. Compared with the time needed for a migration (Section 6.3.3) this is a very long duration.

Oscillation Overlap Model. We also monitored the load of Linux machines. First we are looking at the total free capacity of the cluster. A computer with a load value smaller

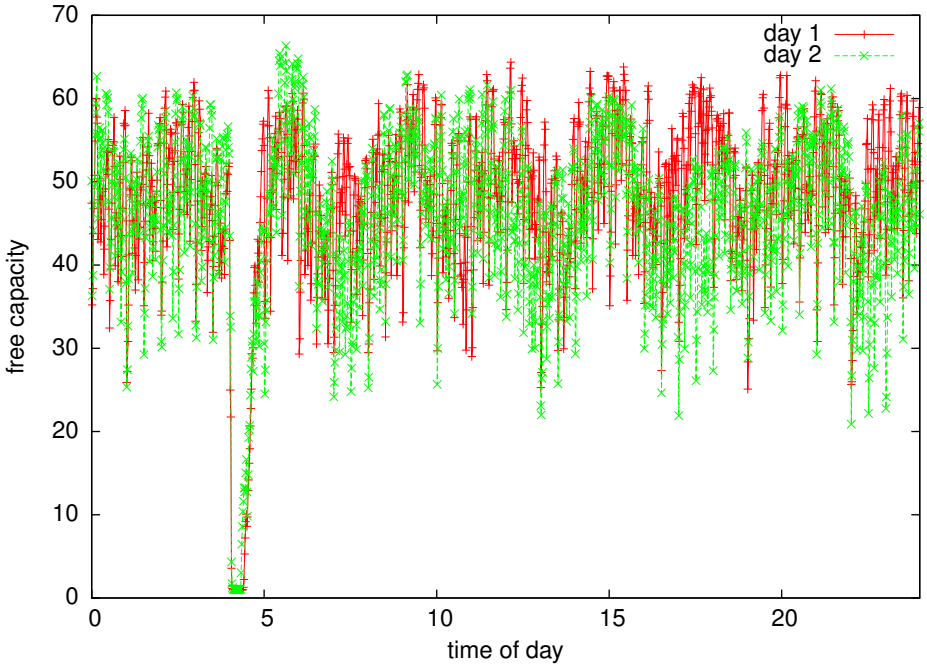


Figure 6.5: Free Resources over the Day

than 1 is not fully utilized, thus it can execute processes of lower priority. So we define the *total free capacity* as the sum of all $1 - l_i$ for all machines i with a load l_i smaller than 1:

$$\text{total free capacity} := \sum_{\substack{i=0,\ldots,p-1 \\ l_i<1}} 1 - l_i$$

A sample of two days can be seen in Figure 6.5. The free capacity falls down every night at four o'clock, when system jobs like checks for software updates are performed on almost all hosts.

Looking at the measured data, one can see that there are some periodic fluctuations, for example, the load depends on the time of the day or the day of the week. Thus, the load of a computer can be seen as an overlapping of oscillations with different frequencies. To analyze the load data, a fast fourier transformation was used to calculate the amplitudes of the frequencies.

Figure 6.6 shows the spectrum of the free capacity. There are peaks at certain frequencies, for instance, 10 min, 15 min, and multiple of them. System jobs are the main reason for this. The operation system checks every 15 minutes if there are some jobs to do, thus the load introduced by the system will always start at the beginning of 15 min time intervals.

The spectrum for a single computer is not that regular, because normally system jobs

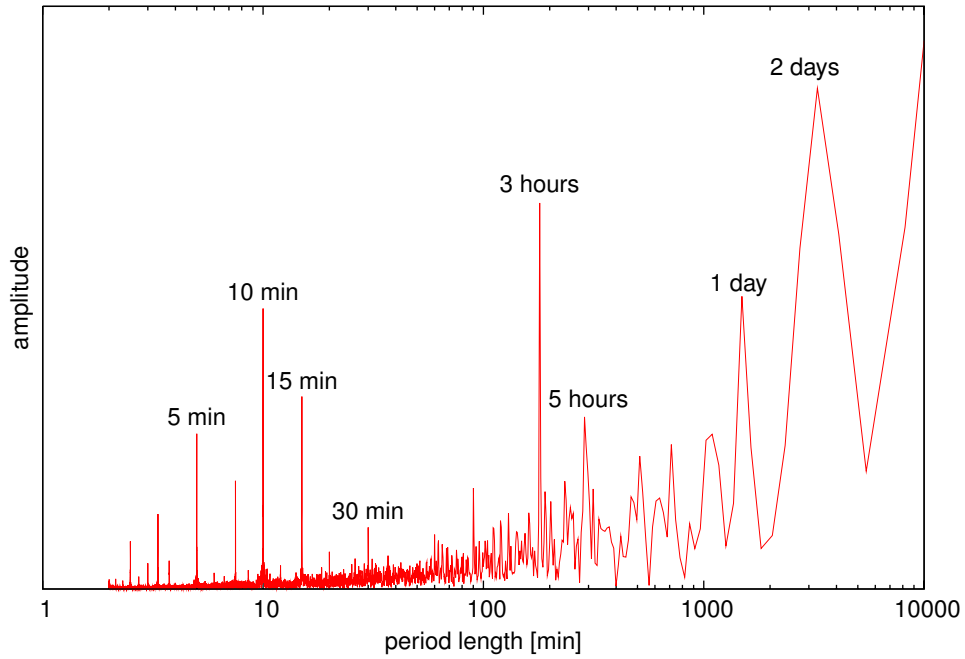


Figure 6.6: Spectrum of Total Free Capacity

do not dominate the work of a computer. Only if one look at the total system, regular parts can be seen, because load differences from the irregular normal work is smoothed out.

6.4.3 Load Balancing Strategies

Load balancing consists of three decisions:

1. Do we need to migrate a virtual processor?
2. To which destination node?
3. Which virtual processor?

Our PUB library contains some loadbalancing strategies, but it is also possible to add new strategies for the first and second question. The virtual processor for a migration is always chosen by PUB: When the load balancer decides to migrate a virtual processor, it writes the destination to a queue and PUB will migrate the first virtual processor that checks this queue, i. e., the first executed on this computer that reaches the synchronization.

In the following, we describe the implemented strategies: one centralized one with one node gathering the load of all nodes and making all migration decisions, and some distributed strategies without global knowledge.

The Global Strategy. All nodes send information about their CPU speed and their actual load to one master node. This node calculates the least ($P_{\min}$) and the most ($P_{\max}$) loaded node and migrates one virtual processor from $P_{\max}$ to $P_{\min}$ if necessary.

Simple Distributed Strategy. Every node asks a constant number c other nodes chosen uniformly at random for their load. If the minimal load of all the c nodes is smaller than the own load minus a constant $d \geq 1$, one process is migrated to the least loaded node. The waiting time between these load balancing messages is chosen uniformly at random in the interval [1 s–40 s] to minimize the probability that two nodes choose the same destination at exactly the same time. If a node is chosen, it will be notified by a message and increases the load value that it reports by one to indicate that the actual load will increase soon.

Conservative Distributed Strategy. As in the simple distributed strategy, c randomly chosen nodes are asked for their load l_i and for their computation power c_i. Next we compute the expected computation power per process $\frac{c_i}{l_i+1}$ if we would migrate one process to the node i for $i = 1, \ldots, c$. If the maximal ratio of all c nodes is greater than the local ratio, the program would run faster after the migration. This strategy only migrates a process, if the source is the most loaded node of the $c+1$ ones, i. e., $\frac{c_i}{l_i}$ is greater than the own ratio. This conservative strategy gives other more loaded nodes the chance to migrate and thus leads to fewer migrations than the simple strategy.

Global Prediction Strategy. Each node has an array of the loads of all other nodes but these entries are not up to date all the time. Each node sends its load periodically to k uniformly at random chosen nodes in the network. In [HDL98] Hill et al. analyze this process using Marcov chains. The mean age of an entry in the array is $\mathcal{O}(p/k)$.

All these strategies use the one minute load average increased by one for each expected task. This is needed because migrations are executed at the synchronization only, and an unloaded machine should not be chosen by many other nodes without noticing that other processes will migrate to the same destination.

6.4.4 Evaluation

Hardware

For our experiments, we used the PCs in our office and in the students' pools. We ran different experiments: During the first experiment, the computers were partially used by their owners, for the second experiment, we choose unused computers and simulated the external workload. For this, we generate artificial load using a higher scheduling priority than our normal jobs. As the simulated work is the same for all tests, we get repeatable results.

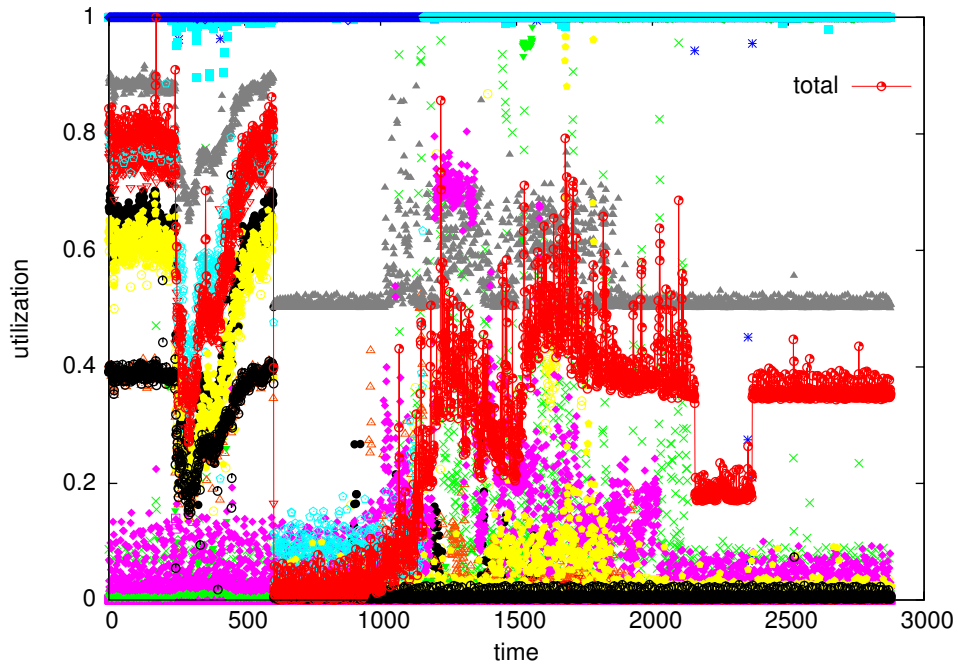


Figure 6.7: Simulated Load Profile

There are computers of different speed:

1. For the first test we use Intel Pentium III with 933 MHz and one or two processors (each processor 1867 bogomips[4]), Intel Pentium IV with 1.7 GHz (3395 bogomips), and 3.0 GHz (5985 bogomips).

2. For the second group, we use 16 workstations powered by Intel Core2 6700 CPUs with 2.66 GHz (each core 5320 bogomips), 4 Intel PentiumD 3 GHz (each core 5990 bogomips), 4 Intel Pentium IV 2.4 GHz (4790 bogomips), and 8 Intel Pentium IV 1.7 GHz (3400 bogomips). It is obvious that bogomips is not an accurate approximation of the performance, because for most applications a Core2 CPU is much faster than a Pentium D. The bogomips dimension depends on clock speed. It is nearly the clock speed or twice the clock speed, if the processor can execute two instructions per cycle. Other important aspects, for example, the speed of the memory hierarchy, are not included.

The load profile to simulate the external load is based on measurements of the utilization of workstation computers. We used the same load profiles as in [BGM05]. One load

[4]bogomips is a simple measuring unit for the speed of CPUs provided by the Linux kernel. We show this unit here because our loadbalancing algorithms use it for approximating the power of the processors.

profile is shown in Figure 6.7. Each entry represents the load of one host, the red curve aggregates the total utilization of the workstations.

Benchmarks

We use different benchmarks for the evaluation. The start with two synthetic microbenchmarks, one for the communication (total exchange, all processors send data to all others), and one for computation (sequential multiplications).

The next two tests are applications: solving the 3-Satisfiability problem and computation of LU-decompositions. We choose these examples as they cover a wide range of typical problems in the field of high performance computations.

Communication Test: Total Exchange. The first example is a simple all-to-all-communication. A run of the benchmark consists of 64 supersteps. In each superstep, each processor send 32 kByte to each other processor. Thus, the cost model of BSP is

$$64 \cdot (32768(p-1) \cdot g + L) = 65,011,712 \cdot g + 64 \cdot L.$$

In an all-to-all comunication, the network is the bottle-neck, so we have to minimize the number of forwarded messages to reduce the network congestion. Figure 6.8 shows the results for the communication benchmark all-to-all on 32 nodes. Updating the routing tables by broadcasting migrations is necessary to reduce the running time significantly.

Artificial Computation Benchmark. This benchmark is a simple computation test. Each host calculates an increasing number of multiplications (from 10^7 up to 10^{11}). Figure 6.9 shows that migrations are profitable for large supersteps. The total execution time of our benchmark decreases from ≈ 8 hours to ≈ 4 hours with migrations.

3-Satisfiability. Our first example application is the same BSP algorithm for the 3-Satisfiability(3-SAT) problem we used in Section 5.5.2 for the system on a chip. We can use the same source code for both systems. This shows that it is easy to implement parallel algorithms that run efficient for very different architectures. In fact, we have 888a few target depending lines of codes, for the input (formula is either read from disk or hard coded into the program) or for time measurements (using either operating system clock or cycle counter).

For the PUB-library implementation the processors enter a new superstep every 10000 divide-steps. If a solution is found, it is broadcasted and the computation stops at the end of the actual superstep. On the one hand these synchronizations are necessary to detect whether a solution is found, on the other hand they are needed because migrations only occur during synchronizations.

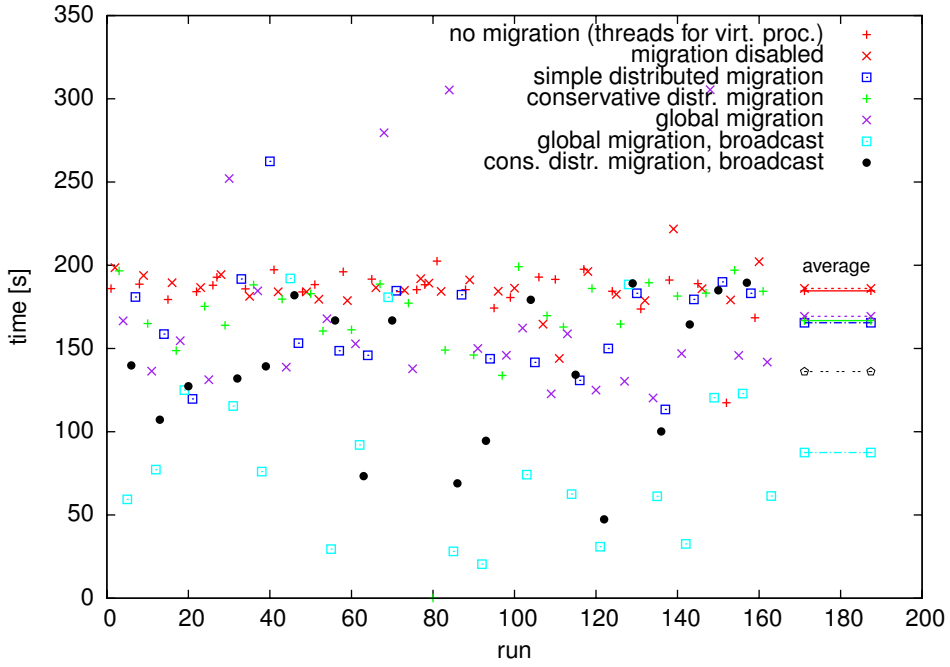


Figure 6.8: Execution Time of Total Exchange on 32 Workstations

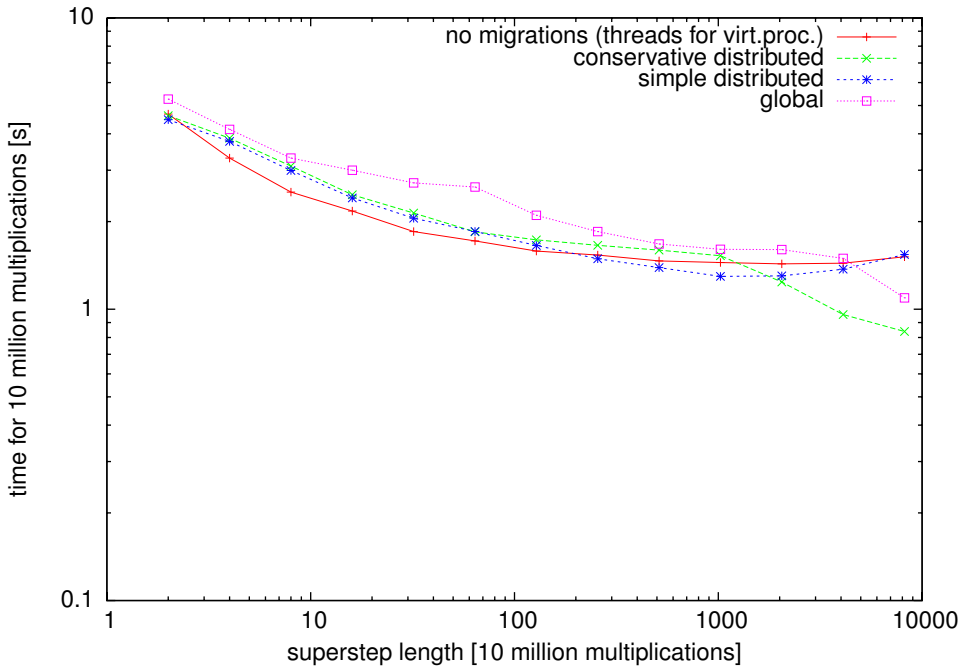


Figure 6.9: Artificial Computation Benchmark

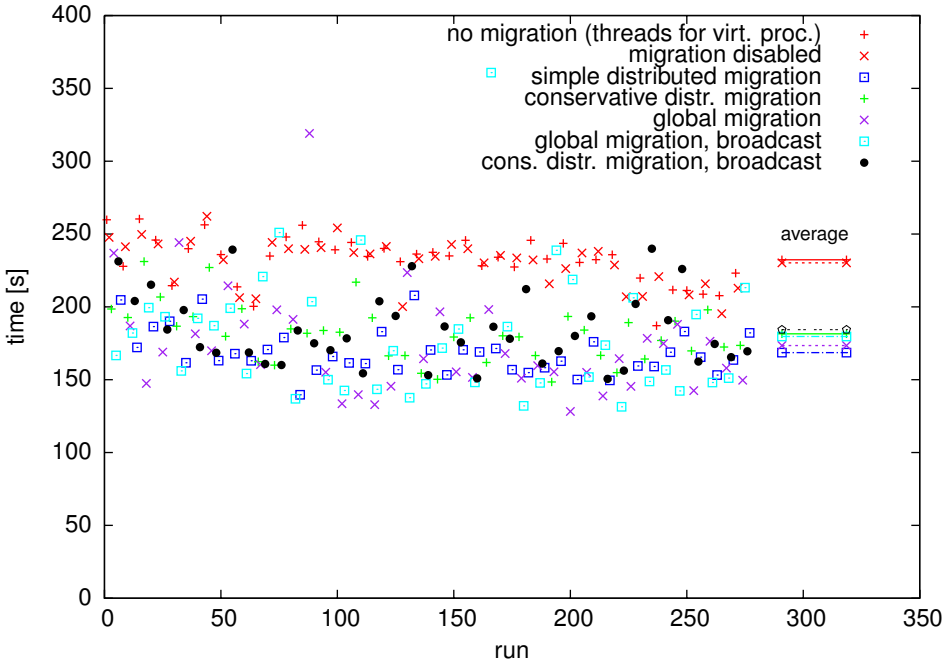


Figure 6.10: Execution Time of 3-SAT on 32 Workstations

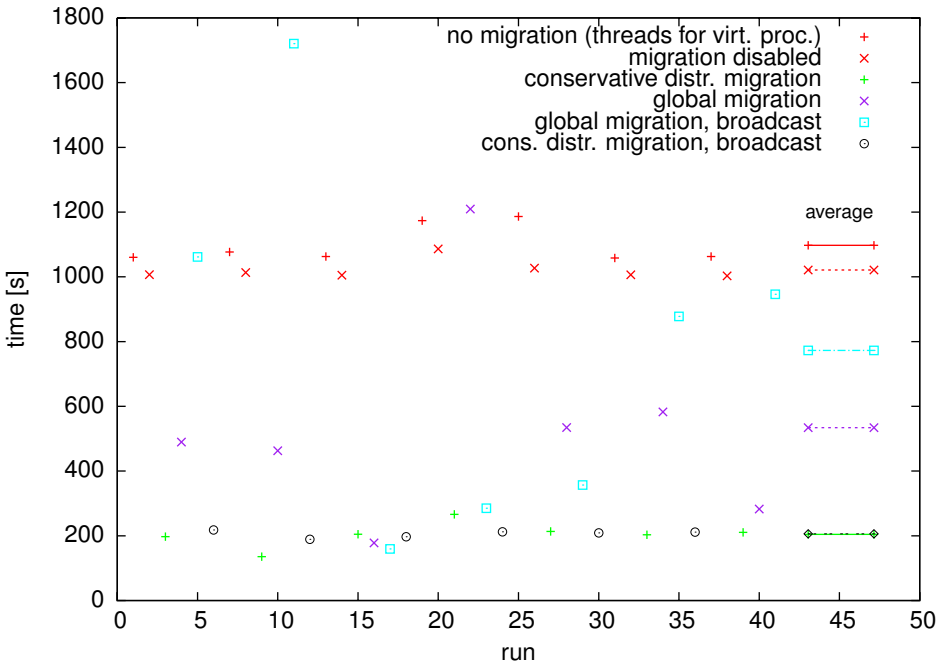


Figure 6.11: Execution Time of 3-SAT on 64 Workstations

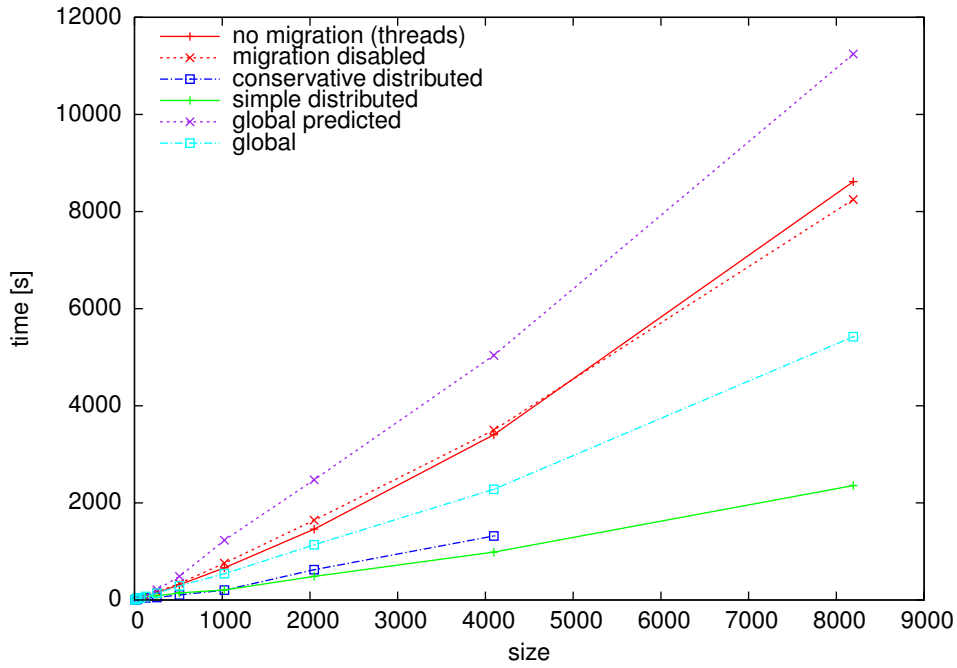
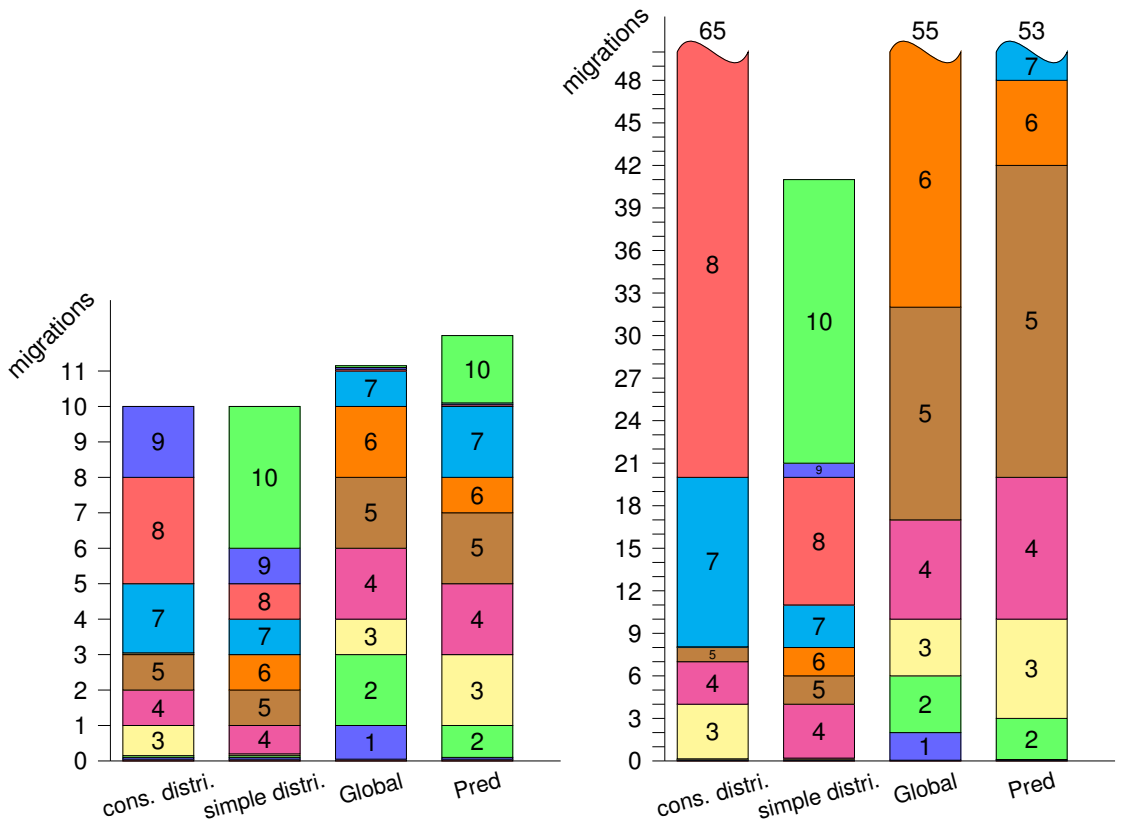


Figure 6.12: LU Decomposition Benchmark

Results. The running time for 3-SAT on 32 and 64 processors is shown in Figures 6.10 and 6.11. The 3-SAT solver benefits from migrations in our office cluster as expected. There were some nodes in the system with some load of other users greater than 1, so it was reasonable to migrate the processes from such nodes to other idle nodes. The cost of the migrations itself is small due to the small memory contexts of 3-SAT. Updating the routing tables is not of big importance, because there are only few messages.

LU Decomposition. A basic problem in many scientific computing applications is solving large systems of linear equations. Therefore, we use a benchmark that deals with matrices and floating point operations. Consider a system of linear equations $Ax = b$, where A is a given $n \times n$ nonsingular matrix, b a given vector of length n, and x is the solution. *LU decomposition* is one method for the calculation of x. LU decomposition is the decomposition of the matrix A into a product of two triangular matrices L and U, $A = L \cdot U$, where L is a unit lower triangular matrix (i. e., ones on the diagonal, zeros above) and U is an upper triangular matrix (i. e., zeros below the diagonal). The advantage of this method over Gaussian elimination is that the factors L and U can be reused for different systems with the same matrix A.

The algorithm and the implementation is described in [Bis04]. The BSP function calls are adapted from the BSPlib syntax to the PUB library syntax, but no further optimization



(a) maximal migrations per host (b) total

Figure 6.13: Number of Migrations

is done, i. e., the benchmark code does not use advanced features of PUB, for example, oblivious synchronization or the adaptive broadcast function.

Results. The LU benchmark benefits from migrations. All load balancing strategies but the global predicted one reduces the total execution times. The distributed strategies perform better than the global one. In our experiment, there was no bottleneck at processor 0, which gathers all load information, but the load information is not up to date all the time. On the other side, the distributed strategies search for less loaded hosts and use only the actual load of the machines. Furthermore, they have a high probability to find imbalances, as the number of machines is only four times the number of hosts asked for their load in a load balancing step.

The number of migrations is shown in Figure 6.13. The simple distributed strategy leads to at most one migration per host and superstep (except the last superstep), whereas

all other strategies have at least 3 supersteps with a maximum of at least 2 migrations for a host. Also the total number of migrations, i. e., the total amount of data sent is significantly lower for the simple distribution strategy (Figure 6.13(b)).

6.5 Conclusions

In this chapter we have shown, how BSP programs can be executed efficiently on clusters of workstations. As we do only use idle times which are wasted in many installations, this is a cheap way to high performance computing. We have demonstrated the power of the concept of virtual processors and the benefits of load balancing by migrations.

The technique of migrating Linux processes in userspace is of independent interest in all areas of load balancing and fault tolerance.

Chapter 7

Summary and Outlook

In this chapter we will summarize the thesis and conclude with some open questions for further research.

7.1 Summary

To provide more performance, computers have to utilize more and more processing units. A general model for such systems is needed, which allows portable yet efficient parallel programs. In this thesis, we evaluated a family of models — the bulk synchronous parallel models — and showed that these models can play the required role. We surveyed algorithms for BSP, and examined implementations of algorithms as well as libraries supporting the implementation of BSP algorithms on different architectures.

The infrequent use of BSP programming in practice is mainly due to the lack of information, hence we hope that this thesis can help to spread knowledge about BSP, and, providing a powerful library, increase the use for real applications.

We proposed a general implementation of BSP, the Paderborn University BSP(PUB) library, which allows the implementation of efficient BSP algorithms for a wide range of architectures in a very comfortable way. In particular, we consider the techniques to utilize unused computational power of existing workstation clusters, as we think much resources could be used better.

We also investigated parallelism on a single chip, as multi core processors are the most promising way to increase performance for every-day user, and in the near future most of the standard computers will be multi core machines. We showed that implementing many simple processors increases parallelism more effectively than implementing only one to four more complex "classical" processors. The benchmarks, although using very small and simple 32 bit RISC processor cores, give an impression of the potential performance of such systems.

We showed that BSP can be useful in such a scenario as well. Thus, BSP algorithms

can be implemented on a large range of parallel systems by using the appropriate compiler and library, without modifying the source code.

7.2 Open Questions and Further Research

Although this thesis gives a complete overview over BSP, from the model to implementations, there are many open questions and links for further research.

Models. There are many extensions for the standard BSP model, but some fields are worthwile to look at, for example, real time applications. BSP can be used to garantuee worst case execution times, but it could likely be improved to suit the needs of real time applications, for instance, by implementing forced synchronizations.

Algorithms. There are many algorithms designed and analysed for the BSP model, but many more problems remain to be treated from the BSP point of view.

Implementations. Implementations of BSP for a large range of parallel machines exist, but as new computer designs appear, one has to implement and optimize the libraries for new technologies, for instance, multi core processors.

Furthermore, more implementations of algorithms, tools to ease the development (including parallel debuggers and performance analysers) are needed. Also, standard libraries of often used data structures are important for the success of a system (e. g., compare Java and its comfortable runtime environment). Hence, one should also consider data structures and implement libraries of standard data structures with a bulk synchronous semantic, for example, distributed containers, including arrays and sorted lists. These containers should have methods to add and delete elements, but these actions should be executed in the synchronization phase only.

Parallel System on Chip. To obtain a high performance system, more powerful processor cores should be used. It would be very interesting to see a hardware implementation employing our interconnection network, fast processor cores (e. g., PowerPC core), and on-chip memory.

We think such a system could outperform standard processors in many applications.

Heterogeneous Workstation Cluster. Several aspects can be considered: First, the time required for migrations can be improved, by “migrating on demand”, i. e., only a small part of the memory is transferred to the destination, then the process is continued, and the migration takes part in the background. If the process accesses a memory page that is not already transmitted, it is suspended until the page is ready.

We have implemented some load balancing algorithms and evaluted their performance in practice. However, there is almost no theoretical analysis of them. One result for load balancing and scheduling in a network is [LMSM04]. Yet, the utilized model differs strongly from our scenario. The authors assume no preemption, and machines are only completely available or not at all. A machine that becomes unavailable terminates all jobs running on it, hence, one has no chance to migrate and save the work done so far.

Our implementation supports fault tolerance. It can save checkpoints of processes to disk and distribute them in the network. The techniques are ready to use, however, a good strategy is missing. At which synchronizations should a checkpoint be created? On which computers should it be stored? Furthermore, a distribution of checkpoint data using idle times of the network should be considered.

Bibliography

[ACS89] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. On communication latency in PRAM computations. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 11–21, Santa Fe, New Mexico, United States, 1989. ACM Press.

[ACS90] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Communication complexity of PRAMs. *Theor. Comput. Sci.*, 71(1):3–28, 1990.

[ADJ+98] Micah Adler, Wolfgang Dittrich, Ben H. H. Juurlink, Mirosław Kutyłowski, and Ingo Rieping. Communication-optimal parallel minimum spanning tree algorithms (extended abstract). In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 27–36, Puerto Vallarta, Mexico, 1998. ACM Press.

[AMD06] *AMD64 Architecture Programmer's Manual, vol. 2: System Programming*. Advanced Micro Devices, Inc., 2006.

[ARM99] ARM Ltd. *AMBA Specification (Rev. 2.0)*, 1999.

[ARM01] ARM Ltd. *Multi-layer AHB*, 2001.

[Bar00] Moshe Bar. The Linux Signals Handling Model. *Linux Journal*, 2000. http://www.linuxjournal.com/article/3985.

[BDM98] Armin Bäumker, Wolfgang Dittrich, and Friedhelm Meyer auf der Heide. Truly Efficient Parallel Algorithms: 1-optimal Multisearch for an Extension of the BSP Model. *Theoretical Computer Science*, 203(2):175–203, 1998.

[Ber98] Martin Beran. Computational power of BSP computers. In *SOFSEM '98: Proceedings of the 25th Conference on Current Trends in Theory and Practice of Informatics*, pages 285–293, London, UK, 1998. Springer-Verlag.

[Ber99] Martin Beran. Decomposable bulk synchronous parallel computers. In *SOFSEM '99: Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*, pages 349–359, London, UK, 1999. Springer-Verlag.

[BGB+04] James M. Baker, Jr., Brian Gold, Mark Bucciero, Sidney Bennett, Rajneesh Mahajan, Priyadarshini Ramachandran, and Jignesh Shah. SCMP: a single-chip message-passing parallel computer. *Journal of Supercomputing*, 30(2):133–149, 2004.

[BGM05] Olaf Bonorden, Joachim Gehweiler, and Friedhelm Meyer auf der Heide. Load balancing strategies in a web computing environment. In *Proceeedings of International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pages 839–846, Poznan, Poland, September 2005.

[BGM06] Olaf Bonorden, Joachim Gehweiler, and Friedhelm Meyer auf der Heide. A web computing environment for parallel algorithms in Java. *Scalable Computing: Practice and Experience*, 7(2):1–14, June 2006.

[BGMZ97] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Marco Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 08(9):943–958, 1997.

[BGOW07] Olaf Bonorden, Joachim Gehweiler, Pawel Olszta, and Rolf Wanka. PUB-Library - User Guide and Function Reference.
Available at http://publibray.sourceforge.net, 2007.

[BGvN89] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument (1946). *Perspectives on the computer revolution*, pages 39–48, 1989.

[Bis04] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, Oxford, UK, March 2004.

[Bis07] Rob H. Bisseling. BSPedupack, source of implementations from [Bis04]. http://www.math.uu.nl/people/bisseling/software.html, 2007.

[BJvR03] Olaf Bonorden, Ben H. H. Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, February 2003.

[BL98] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4–5):361–372, March 1998.

[BMW02] Olaf Bonorden, Friedhelm Meyer auf der Heide, and Rolf Wanka. Composition of efficient nested BSP algorithms: Minimum spanning tree computation as an instructive example. In *Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2002.

[Bon02] Olaf Bonorden. Ein System zur automatischen Konfiguration effizienter paralleler Algorithmen im BSP-Modell. Diplomarbeit (in German), Universität Paderborn, 2002.

[Bon07] Olaf Bonorden. Load balancing in the bulk-synchronous-parallel setting using process migrations. In *The Sixteenth International Heterogeneity in Computing Workshop (HCW07), Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS07)*. IEEE Computer Society, 2007.

[BR00] Stefan Bock and Otto Rosenberg. A new parallel breadth first tabu search technique for solving production planning problems. *International Transactions in Operational Research*, 7(6):625–635, 2000.

[BS97] David Blackston and Torsten Suel. Highly portable and efficient implementations of parallel adaptive n-body methods. In *SC '97: High Performance Networking and Computing (Supercomputing '97)*, November 1997.

[BSP07] BSP Worldwide, Co-ordinating Bulk Synchronous Parallal Computation. http://www.bsp-worldwide.org/, 2007. Rob H. Bisseling (maintainer).

[BSR+03] Olaf Bonorden, Adrian Slowik, Ulrich Rückert, Mario Porrmann, Jörg-Christian Niemann, Friedhelm Meyer auf der Heide, Uwe Kastens, Dinh Khoi Le, Nikolaus Brüls, and Michael Thies. A holistic methodology for network processor design. In *Proceedings of the Workshop on High-Speed Local Networks held in conjunction with the 28th Annual IEEE Conference on Local Computer Networks (LCN2003)*, pages 583–592, October 2003.

[BvzGG+01] Olaf Bonorden, Joachim von zur Gathen, Jürgen Gerhard, Olaf Müller, and Michael Nöcker. Factoring a binary polynomial of degree over one million. *ACM SIGSAM Bulletin*, 35(1):16–18, 2001.

[Cal96] Radu Calinescu. Bulk Synchronous Parallel Algorithms for Optimistic Discrete Event Simulation. Technical Report PRG-TR-8-9, Oxford University Computing Laboratory, April 1996.

[CKP+93] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[CLG05] Jiannong Cao, Yinghao Li, and Minyi Guo. Process migration for MPI applications based on coordinated checkpoint. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems*, volume 1, pages 306–312, 2005.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1990.

[CP93] Robert Cypher and C. Greg Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *Journal of Computer and System Science*, 47(3):501–548, 1993.

[DFRC93] Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *SCG '93: Proceedings of the ninth annual symposium on Computational geometry*, pages 298–307, New York, NY, USA, 1993. ACM Press.

[DFRC96] Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal of Computational Geometry and Applications*, 6(3):379–400, 1996.

[DG98] Frank Dehne and Silvia Götz. Practical parallel algorithms for minimum spanning trees. In *Proceedings 17th IEEE Symposium on Reliable Distributed Systems, Workshop on Advances in Parallel and Distributed Systems, West Lafayette, IN, USA*, pages 366–371, 1998.

[DHS97] Stephen R. Donaldson, Jonathan M. D. Hill, and David B. Skillicorn. Predictable communication on unpredictable networks: Implementing BSP over TCP/IP. Technical Report PRG-TR-40-, Programming Research Group, Oxford University Computing Laboratory, 1997.

[dis08] distributed.net project homepage. http://www.distributed.net, 2008.

[DL01] William R. Dieter and James E. Lumpp, Jr. User-level checkpointing for linuxthreads programs. In *Proceedings of the 2001 USENIX Technical Conference*, http://www.engr.uky.edu/ dieter/publications.html, June 2001.

[DR98] Bozhidar Dimitrov and Vernon Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):459–469, 1998.

[Dre05] Ulrich Drepper. ELF Handling For Thread-Local Storage. http://people.redhat.com/drepper/tls.pdf, December 2005. Version 0.20.

[Due05] Jason Duell. The design and implementation of Berkeley Lab's Linux checkpoint/restart. Technical Report LBNL-54941, April 2005.

[FH96] Amr Fahmy and Abdelsalam Heddaya. Communicable memory and lazy barriers for bulk synchronous parallelism in BSPk. Technical Report BUCS-TR-1996-01, Computer Science Department, Boston University, July 1996.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, New York, NY, USA, 1978. ACM Press.

[GGHK05] Andrei Goldchleger, Alfredo Goldman, Ulisses Hayashida, and Fabio Kon. The implementation of the BSP parallel computing model on the InteGrade Grid middleware. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[GHL+96] Mark W. Goudreau, Jonathan M. D. Hill, Kevin Lang, Bill McColl, Satish B. Rao, Dan C. Stefanescu, Torsten Suel, and Thanasis Tsantilas. A proposal for the BSP Worldwide Standard Library. Technical report, http://www.bsp-worldwide.org, April 1996.

[Gib89] Phillip B. Gibbons. A more practical PRAM model. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168, Santa Fe, New Mexico, United States, 1989. ACM Press.

[GLC01] Yan Gu, Bu-Sung Lee, and Wentong Cai. JBSP: a BSP programming library in Java. *Journal of Parallel and Distributed Computing*, 61(8):1126–1142, 2001.

[GLP+00] Jesus A. Gonzalez, Coromoto Leon, Fabiana Piccoli, Marcela Printista, José Luis Roda, Casiano Rodríguez, and Francisco de Sande. Oblivious bsp. In *EUROPAR: Parallel Processing, 6nd International EURO-PAR Conference*, volume 1900, pages 682–685. Springer, 2000.

[GLR+99] Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, and Thanasis Tsantilas. Portable and efficient parallel computing using the BSP model. *IEEE Transactions on Computers*, 48(7):670–689, 1999.

[GMR98] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, 28(2):733–769, 1998.

[Goo00] Michael T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 2000.

[Göt98] Silvia Götz. Communication-efficient parallel algorithms for minimal spanning tree computations. Diploma Thesis, Universität Paderborn, 1998.

[Gun04] Neil J. Gunther. Linux load average revealed. In *Proceedings of the 30th International Computer Measurement Group (CMG) Conference*, pages 149–160, December 2004.

[GV94] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel Distributed Computing*, 22(2):251–267, 1994.

[Hay02] Mike Hayward. Intel P6 vs P7 system call performance. http://lkml.org/lkml/2002/12/9/13, 2002. LWN.net.

[HDL98] Jonathan M. D. Hill, Stephen R. Donaldson, and Tim Lanfear. Process migration and fault tolerance of BSPlib programs running on networks of workstations. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, pages 80–91, London, UK, 1998. Springer-Verlag.

[HDS97] Jonathan M. D. Hill, Stephen R. Donaldson, and David B. Skillicorn. Stability of Communication Performance in Practice: from the Cray T3E to Networks of Workstations. Technical Report PRG-TR-33-97, Programming Research Group, Oxford University Computing Laboratory, 1997.

[Hil98] Jonathan M. D. Hill. Oxford BSP toolset. http://www.bsp-worldwide.org/implmnts/oxtool/, 1998.

[HJSV98] Jonathan M.D. Hill, Stephen Jarvis, Constantinos J. Siniolakis, and Vasil P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*, page 0286, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[HL02] Gaétan Hains and Frédéric Loulergue. Functional bulk synchronous parallel programming using the BSMLlib Library. In Sergei Gorlatch and Christian Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, pages 165–178. Nova Science Books and Journals, 2002.

[HMS+98] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[HO01] Juraj Hromkovič and Waldyr M. Oliva. *Algorithmics for Hard Problems*. Springer-Verlag New York, Inc., 2001.

[HS00] Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. In Ian P. Gent, Hans Van Maaren, and Toby Walsh, editors, *Sat 2000: Highlights of Satisfiability Research in the Year 2000*, volume 63 of *Frontiers in Artificial Intelligence and Applications*, pages 283 – 292. IOS Press, 2000.

[IEE96] IEEE. *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]*. IEEE, New York, NY, USA, 1996.

[IEE05] IEEE. 1666-2005 IEEE Standard System C Language Reference Manual. http://standards.ieee.org/getieee/1666/index.html, 2005.

[Int06] Intel Corporation. Press Kit — Moore's Law 40th Anniversary. http://www.intel.com/pressroom/kits/events/moores_law_40th/index.htm, 2006.

[ISO05] *ISO/IEC 9899:1999: Programming languages – C*. International Organization for Standardization, Geneva, Switzerland, 2005.

[JKMR03] Ben H. H. Juurlink, Petr Kolman, Friedhelm Meyer auf der Heide, and Ingo Rieping. Optimal broadcast on parallel locality models. *Journal of Discrete Algorithms*, 1(2):151–166, 2003.

[Juu97] Ben H. H. Juurlink. *Computational Models for Parallel Computers*. Dissertation, Leiden University, Utrecht, 1997.

[JW96] Ben H. H. Juurlink and Harry A. G. Wijshoff. The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model.

In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, pages 339–347, London, UK, 1996. Springer-Verlag.

[KDH+05] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4-5):589–604, 2005.

[KeN06] Kernel Newbies: Linux 2.6.18. http://kernelnewbies.org/Linux_2_6_18, 2006.

[Keß00] Christoph W. Keßler. NestStep: Nested parallelism and virtual shared memory for the BSP Model. *The Journal of Supercomputing*, 17(3):245–262, November 2000.

[KLST04] Uwe Kastens, Dinh Khoi Le, Adrian Slowik, and Michael Thies. Feedback driven instruction-set extension. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 126–135, New York, NY, USA, 2004. ACM Press.

[Kun91] Thomas Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, 1991.

[LMR95] Zhiyong Li, Peter H. Mills, and John H. Reif. Models and resource metrics for parallel and distributed computation. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, pages 51–60, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[LMSM04] Stefano Leonardi, Alberto Marchetti-Spaccamela, and Friedhelm Meyer auf der Heide. Scheduling against an adversarial network. In *Proc. 16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2004)*, 2004.

[LTBL97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison Computer Sciences, April 1997.

[LWN05] Address space randomization in 2.6. http://lwn.net/Articles/121845/, 2005. LWN.net.

[McC95a] William F. McColl. A BSP realisation of Strassen's algorithm. Technical report, Oxford University Computing Laboratory, May 1995.

[McC95b] William F. McColl. Scalable Computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000, pages 46–61. Springer-Verlag, 1995.

[MGLV04] Christine Morin, Pascal Gallard, Renaud Lottiaux, and Geoffroy Vallée. Towards an efficient single system image cluster operating system. *Future Gener. Comput. Syst.*, 20(4):505–521, 2004.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38:114–117, April 1965.

[Moo98] Gordon E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. Reprint of Moo65.

[Mot98a] Motorola. *M-Core Reference Manual*, 1998.

[Mot98b] Motorola. *MMC2001 Reference Manual*, 1998.

[MPI95] MPI Forum. MPI: A Message Passing Interface Standard. http://www.mpi-forum.org, June 1995.

[MT99] William F. McColl and Alexandre Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, 24(3-4):287–297, August 1999.

[MT04] Jeremy Martin and Alexandre Tiskin. Dynamic BSP: Towards a Flexible Approach to Parallel Computing over the Grid. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 219–226, 2004.

[MW00] Jeremy Martin and Alex Wilson. A visual BSP programming environment for distributed computing. In *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 15–29, 2000.

[Nag04] Norman Nagel. Analyse eines massiv parallelen Multiprozessorsystems mit SystemC. Diploma thesis, in german, University of Paderborn, 2004.

[NLPR07] Jörg-Christian Niemann, Christian Liß, Mario Porrmann, and Ulrich Rückert. In *ARCS'07: Architecture of Computing Systems*, pages 83–97, Zurich, Switzerland, 2007.

[Nöc01] Michael Nöcker. *Data structures for parallel exponentiation in finite fields*. Doktorarbeit, Universität Paderborn, Germany, Juni 2001.

[NPPR07] Jörg-Christian Niemann, Christoph Puttmann, Mario Porrmann, and Ulrich Rückert. Resource efficiency of the giganetic chip multiprocessor architecture. *Journal of Systems Architecture*, 53(5-6):285–299, 2007.

[NPSR05] Jörg-Christian Niemann, Mario Porrmann, Christian Sauer, and Ulrich Rückert. An evaluation of the scalable giganetic architecture for access networks. In *Advanced Networking and Communications Hardware Workshop (ANCHOR), held in conjunction with the 32nd Annual International Symposium on Computer Architecture (ISCA 2005)*, 2005.

[OIS+06] Moriyoshi Ohara, Hiroshi Inoue, Yukihiko Sohda, Hideaki Komatsu, and Toshio Nakatani. MPI microtask for programming the cell broadband engine^{tm} processor. *IBM Systems Journal*, 45(1):85–102, 2006.

[PNPR07] Christoph Puttmann, Jörg-Christian Niemann, Mario Porrmann, and Ulrich Rückert. Giganoc – a hierarchical network-on-chip for scalable chip-multiprocessors. In *Proceedings of the 10th EUROMICRO Conference on Digital System Design (DSD)*, pages 495–502, 2007.

[pub08] Paderborn University BSP Library. http://publibrary.sourceforge.net, 2008.

[PW72] William Wesley Peterson and E. J. Weldon, Jr. *Error-Correcting Codes*. MIT Press, Cambridge, Massachusetts, 2nd edition, 1972.

[Rie00] Ingo Rieping. *Communication in Parallel Systems–Models, Algorithms and Implementations*. Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Theoretische Informatik, 2000.

[Sar98] Luis F. G. Sarmenta. Bayanihan: Web-based volunteer computing using Java. In *WWCA '98: Proceedings of the Second International Conference on Worldwide Computing and Its Applications*, pages 444–461, London, UK, 1998. Springer-Verlag.

[Sar99] Luis F. G. Sarmenta. An adaptive, fault-tolerant implementation of BSP for JAVA-based volunteer computing systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 763–780, London, UK, 1999. Springer-Verlag.

[SAT00] SATLIB – Benchmark Problems. http://www.cs.ubc.ca/ hoos/SATLIB/benchm.html, 2000. Holger H. Hoos, maintainer, The University of British Columbia.

[SB05] Sven Schneider and Robert Baumgartl. Unintrusively measuring Linux kernel execution times. In *7th RTL Workshop*, pages 1–5, November 2005.

[set08] SETI@Home project homepage. http://setiathome.ssl.berkeley.edu, 2008.

[SHM96] David B. Skillicorn, Jonathan M. D. Hill, and William F. McColl. Questions and answers about BSP. Technical Report TR-15-96, Oxford University Computing Laboratory, 1996.

[SS63] John C. Shepherdson and Howard E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2):217–255, 1963.

[SS92] Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel Distributed Computing*, 14(4):361–372, 1992.

[SSB+05] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[SSY00] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode transformation for portable thread migration in Java. Technical report, University of Tokyo, 2000.

[ST98] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, 30(2):123–169, 1998.

[Sta81] Richard M. Stallman. Emacs the extensible, customizable self-documenting display editor. *SIGPLAN Notices*, 16(6):147–156, 1981.

[Str06] Erich Strohmaier. TOP500—TOP500 supercomputer. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 18, New York, NY, USA, 2006. ACM Press.

[Sui07] Wijnand J. Suijlen. BSPonMPI. http://bsponmpi.sourceforge.net, January 2007, 2007.

[sys07] Open SystemC Initiative. http://www.systemc.org, 2007.

[Tis01] Alexandre Tiskin. All-pairs shortest paths computation in the BSP model. In P. G. Spirakis F. Orejas and J. van Leeuwen, editors, *Proceedings of ICALP*, volume 2076, pages 178–189, 2001.

[Tis03] Alexandre Tiskin. Communication-efficient parallel gaussian elimination. In V. Malyshkin, editor, *Proceedings of PaCT*, volume 2763, pages 369–383, 2003.

[Tis07] Alexandre Tiskin. Communication-efficient parallel generic pairwise elimination. *Future Generation Computer Systems*, 23(2):179–188, 2007.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[vL90] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier and MIT Press, 1990.

[WP00] Tiffani L. Williams and Rebecca J. Parsons. The heterogeneous bulk synchronous parallel model. In *Workshop on Advances in Parallel and Distributed Computational Models (APDCM'00)*, Lecture Notes in Computer Science, pages 102–108, 2000.

[XL97] Hong Xie and Wanqing Li. Parallel volume rendering using the BSP model. In Hongchi Shi and Patrick C. Coffield, editors, *Proceedings of Society of Photo-Optical Instrumentation Engineers (SPIE) Conference, Parallel and Distributed Methods for Image Processing*, volume 3166, pages 274–279, September 1997.

[ZKX99] Weiqun Zheng, Shamim Khan, and Hong Xie. BSP Pro: a Java-based BSP performance profiling system. In *Proceedings of the Fourth International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, pages 54–59, 1999.

Nomenclature

C_1 12
class of machine models, machines that are polynomial-time equivalent and linearly space equivalent to DTM, cf. [vL90].

C_2 12
class of machine models, machines that can all problems of the class $\mathcal{PSPACE}$ in polynomial time, cf. [vL90].

3-SAT 66
3-Satisfiability problem, problem of determining if the variables of a given boolean formula can be assigned.

AMBA 54
ARM advanced microcontroller bus architecture.

Broadcast 19
communication problem, one processor want to send the same data to all others.

BSP 10
bulk synchronous parallel, a parallel computing model.

Cache 53
temporary storage area where frequently accessed data can be stored for rapid access.

CPU 1
central processing unit.

CRC 41
cyclic redundancy check, algorithm to compute a checksum of data.

CRCW 8
concurrent read concurrent write, variant of PRAM allowing concurrent accesses to a memory cell.

EREW 8
exclusive read exclusive write, variant of PRAM not allowing concurrent accesses to a memory cell.

MMU 41
memory management unit, part of the processor that manage the virtual address space.

MPI 36
message passing interface, a standard for communication in parallel systems using messages [MPI95].

MST 29
minimal spanning tree problem, problem of finding a tree connecting all nodes of a graph with minimal weights of the tree edges.

PC 11
personal computer, a computer intended to be operated directly be an end user.

PRAM 8
parallel random access machine, a parallel computing model.

PUB 37
Paderborn University BSP library, a C library for implementation of BSP algorithm.

S-Core 53
32 bit RISC processor core used for the parallel system on chip.

Shared Memoy 51
random access memory that is accessable by different units.

SystemC 57
C++ library to describe and simulate concurrent processes, cf. [sys07].

Total Exchange 19
communication problem, all processors have data for all others.

VHDL 50
very high speed integrated circuit hardware description language.

Index