UNIVERSITY OF LEIPZIG
Faculty of Mathematics and Computer Science
Department of Computer Science

Automatic Parallelization and Minimization of Communication Costs of Program Instances

Diploma Thesis

Diplomarbeit

Supervisor: Prof. Dr. Claudia Leopold

Leipzig, October 30, 2003

presented by
Michael Süß
Date of Birth: October 24, 1978
Study Course: Computer Science

Abstract

Writing a program that is able to use a parallel machine effectively is a challenging and expensive task even today, since many optimizations can only be done by experienced programmers. Although automatically parallelizing compilers are slowly maturing, they are still limited in their scope. This thesis tries to find a solution in between, by adapting an approach called *semi–automatic method for locality optimization* that was first suggested by Leopold in [Leo98]. The method consists of three steps. First, small program instances of a given program are considered, and are then optimized for a fixed number of processors. In a final step, the structure of the solution is generalized to the initial program.

The major contribution of this thesis is a local search algorithm, which automatically parallelizes program instances by distributing statements onto a number of processors. The algorithm is a modified version of the one used in [Leo98], with a completely new objective function based on simulated memory–access times. Experimental results indicate that the modified semi–automatic method is able to optimize program instances for parallel execution, while simultaneously fulfilling statement dependencies. Furthermore, cache locality on individual processors is optimized, too.

Kurzfassung

Ein Programm zu schreiben, welches einen Parallelrechner effektiv nutzt, ist sogar heutzutage noch eine herausfordernde und teure Aufgabe, da viele Optimierungsschritte nur von erfahrenen Programmierern ausgeführt werden können. Obwohl automatisch parallelisierende Compiler bereits einen gewissen Reifegrad erreichen, sind sie in ihren Ansätzen immer noch beschränkt. Diese Diplomarbeit versucht, einen Mittelweg zu finden, indem die von Leopold in [Leo98] vorgeschlagene halb–automatische Methode zur Lokalitätsoptimierung benutzt und angepasst wird. Diese besteht aus drei Schritten: Zuerst wird eine kleine Programminstanz eines gegebenen Programms betrachtet, welche dann für eine feste Anzahl von Prozessoren optimiert wird. In einem letzten Schritt wird die Struktur der gefundenen Lösung auf das ursprüngliche Programm verallgemeinert.

Der Hauptbeitrag dieser Diplomarbeit ist ein lokaler Suchalgorithmus, der automatisch Programminstanzen parallelisiert, indem die Anweisungen auf eine feste Anzahl Prozessoren

verteilt werden. Der Algorithmus ist eine modifizierte Version des in [Leo98] skizzierten, allerdings mit einer komplett neuen Zielfunktion, die auf simulierten Speicherzugriffszeiten basiert. Erste experimentelle Resultate lassen den Schluss zu, dass die angepasste halbautomatische Methode in der Lage ist, Programminstanzen für den Parallelbetrieb zu optimieren, und darüber hinaus bestehende Abhängigkeiten zwischen Anweisungen zu beachten. Zusätzlich wird die Cache–Lokalität auf den einzelnen Prozessoren optimiert.

Contents

List of Figures **v**

List of Tables **vi**

List of Abbreviations **vii**

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Contributions . . . 2
1.3 Structure . . . 2
1.4 Acknowledgments . . . 3
1.5 Miscellaneous . . . 3
1.6 Disclaimer . . . 4

2 Foundations **5**

2.1 The Memory Hierarchy . . . 5
2.1.1 Caches . . . 8
2.2 An Overview of Parallelism . . . 11
2.2.1 Speedup, Efficiency and Amdahl's Law . . . 11
2.2.2 Flynn's Classification . . . 13
2.2.3 Memory Organization in Parallel Architectures . . . 15
2.2.4 Interprocessor Communication . . . 20
2.2.5 An Introduction to `OpenMP` . . . 20
2.2.6 Automatic Parallelization and Dependencies . . . 23
2.3 The Semi–Automatic Method and the `iblOpt` Programming Tool . . . 24

3 Conceptual Work **27**

3.1 The Optimization Algorithm . . . 28
3.2 The Objective Function . . . 33
3.2.1 The Runtime Simulator . . . 35
3.3 Suitable Examples . . . 36

4 Implementation **40**

4.1 Header Files 40
4.2 Program Files 41

5 Results **43**
5.1 Matrix Transposition 43
5.2 Matrix Multiplication 46
5.3 The Laplace Equation 49

6 Conclusions **52**
6.1 Summary 52
6.2 Outlook 52

A Sequential Program Sources **55**
A.1 traposeq.c 55
A.2 multseq.c 57
A.3 laplaceseq.c 58

B Parallel Program Sources **61**
B.1 trapopara.c 61
B.2 multpara.c 63
B.3 laplacepara.c 65

Bibliography **70**

Index **74**

Statement **74**

List of Figures

2.1 A memory hierarchy with 5 levels 6
2.2 Sample speedup representation 12
2.3 Sample efficiency representation 13
2.4 A classification of parallel architectures 14
2.5 A Uniform Memory Access architecture 16
2.6 A Nonuniform Memory Access architecture 17
2.7 Program execution scheme for `OpenMP` programs 21
2.8 A sample `C` program, including `OpenMP` directives 22
2.9 Work sharing constructs in `OpenMP` 22
2.10 A graphical representation of the semi–automatic method 25

3.1 A generic local search algorithm, executed repeatedly to increase the chances of finding global optima 28
3.2 Representation of a cross dependency 29
3.3 The local search algorithm used for automatic parallelization 32
3.4 The wrapper objective function 35
3.5 The runtime simulation function 37

5.1 An optimized PI for matrix transposition with N = 4 on two processors 45
5.2 An optimized PI for matrix multiplication with N = 4 on two processors 48
5.3 An optimized PI for the laplace equation with N = 4 on four processors 50

List of Tables

1.1 Typographical conventions 3

2.1 Reasons for locality 6
2.2 Representations of dependencies 24

5.1 `iblOpt` results for matrix transposition 44
5.2 Timing results for the parallel matrix transposition program 46
5.3 `iblOpt` results for matrix multiplication 47
5.4 Timing results for the parallel matrix multiplication program 49
5.5 `iblOpt` results for the laplace equation 49
5.6 Timing results for the parallel laplace equation program 51

List of Abbreviations

ANSI	American National Standards Institute
API	Application Programming Interface
CHT	Cache Hit Time
CST	Cache Sync Time
DAG	Directed Acyclic Graph
DI	Data Instance
HHLR	Hessischer Hochleistungsrechner
IBLOPT	Instance Based Locality Optimization
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MLT	Memory Load Time
MPI	Message Passing Interface
NUMA	Nonuniform Memory Access
PI	Program Instance
PVM	Parallel Virtual Machine
RAM	Random Access Memory
SI	Statement Instance
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMP	Symmetric Multiprocessor
UMA	Uniform Memory Access

Chapter 1

Introduction

1.1 Motivation

During the last decade, high–performance computing has seen a tremendous increase in speed and efficiency. While faster processors, memory subsystems, storage media and buses have done their part to make this success story possible, the heavy use of *parallelization* was the key to achieving these speedups. And yet there is no end in sight, because the demand of computing power is rising steadily, since problems like weather prediction, geological simulations or calculations of astronomical bodies in space (to name just a few) are using more accurate and therefore numerically expensive models. There are disadvantages and problems to be considered with parallelization though. Sequential programs need to be rewritten, or new ones must be designed, to make use of the available parallel computing power. Usually, this task is accomplished by specialists with years of experience in the field of parallelization. While the introduction of standards like `PVM` or `MPI` has made that work easier, it remains a difficult and time consuming task to write efficient, clean and correct parallel code.

A possible way–out is *automatic parallelization*. Kernighan and Pike only retell an old truth, when they state in one of their landmark publications:

> *Automation is under–appreciated. It is much more effective to have a computer do your work than to do it by hand. We saw examples in testing, in debugging, in performance analysis, and notably in writing code, where for the right problem domain, programs can create programs that would be hard for people to write.* ([KP99, p. 248])

Much research has gone into that field over the last 30 years, and some of it is summarized in section 2.2.6 of this thesis. Various compilers for parallel computers have left their childhood, and are becoming mature. However, there are still limitations for these compilers, and especially in the field of scientific computing much can be gained by hand optimizing an algorithm for parallelism.

Pretty much the same thing can be said about the treatment of locality. There have been no recent breakthroughs in the design of memory hierarchies. Their latencies have not been reduced nearly as much as processor speeds have grown, and therefore in our times the memory subsystems are more of a bottleneck than ever. More levels of cache were not able to solve the problem permanently, but merely brought some relief. For the reasons stated, designing algorithms and compilers with locality in mind is an important, and yet difficult task. Many compilers in use today at least claim to optimize for some kind of locality, by using various techniques out of the field of loop reordering, e. g. *stride–1 indexing* or *tiling*. But still, writing a program, in which the compiler is able to recognize all possible instances of locality remains a challenging task.

With these considerations in mind, the `iblOpt` programming tool has been designed and implemented. It provides a relatively easy way to discover variants of algorithms with a high degree of locality, through the utilization of *program instances*. This thesis aims at enhancing the tool to provide a unified approach to automatic parallelization and locality optimization, while building on the already tested algorithms of the `iblOpt` program.

Like `iblOpt`, the extended tool is embedded in a *semi–automatic method* and processes a performance critical section of a sequential program as its input. This input is transformed into a sequence of *statement instances*, after loop borders and other parameters have been fixed. For automatic parallelization, these statement instances need to be distributed across a fixed number of processors on a `Symmetric Multiprocessor` architecture (`SMP`) with minimal communication costs between nodes, such that data dependencies are honored. A good distribution is reached, when the total execution time of the given program instance is at a minimum. Furthermore, the target function includes caches of a fixed size with faster access times than main memory.

1.2 Contributions

This thesis makes three contributions. First, it provides an exact definition of an objective function for the new scenario, based on the one used in `iblOpt`. Second, it introduces an optimization algorithm on the basis of a local search approach, whose purpose is the distribution of the statement instances across processors. This algorithm has been implemented and tested. Finally, it gives experimental results for several sequential example programs.

1.3 Structure

The thesis consists of six parts. Chapter 2 provides some foundations, which are important to understand the rest of this thesis. Special emphasis is placed on memory hierarchies (in particular caches), basic concepts of parallel processing, and a description of the *semi–automatic*

method including the `iblOpt` programming tool. Of course, not all aspects of these topics can be covered in full detail here, since this would fill several books, so only the ones with a high relevance for this work are examined. Pointers to additional literature have been added for further reading.

The focus of chapter 3 is conceptual work; it explains the objective function, algorithms and test programs (as well as the reasons for choosing them) used to achieve our goals. Some implementation details are given in chapter 4. Experimental results are presented in chapter 5, where we examine a few sequential test programs. The experiments consist of both *program instances* and programs, with the latter being studied based on some common performance measures.

Chapter 6 provides a short summary of this thesis and outlines possible directions for future work, including how the approaches presented here could be ported to different parallel architectures and cache configurations.

1.4 Acknowledgments

First of all I have to thank Prof. Dr. Claudia Leopold for providing and assigning this interesting and challenging topic. Although we were locally separated, she managed to answer my questions promptly and helped me sort out my ideas into usable plans. With her emails and personal encouragements, she provided a productive environment, which I enjoyed much during my thesis time.

I feel fortunate that Prof. Dr. Martin Middendorf has agreed to become a member of my thesis committee. I am in debt to him for asking critical questions, and providing valuable advice as well.

Special thanks goes to my parents. Their silent support enables me to pursue my goals and dreams, no matter whether they are related to education or not.

1.5 Miscellaneous

For readability, different font styles have been used to denote names, terms and identifiers, as shown in table 1.1.

Style	Usage
`type writer`	products, companies, computers, architectures, programs, files, functions, variables, data structures, data types
emphasize	special accentuation
slanted	German translations

Table 1.1: Typographical conventions

1.6 Disclaimer

Trademarks and brand names have been used without explicitly indicating them. The absence of trademark symbols does not infer that a name or a product is not protected. All trademarks are the property of their respective owners.

Chapter 2

Foundations

2.1 The Memory Hierarchy

The memory subsystem has always been considered one of the most performance–critical parts of any computer architecture. A perfect computer would therefore include a huge amount of fast memory. Unfortunately, this is, and probably will always remain, impossible, since the fastest memory available is always the most expensive one, too. Therefore, a slightly different approach is pursued in our days, by effectively using a hierarchy of memories with different speeds and sizes. When this hierarchy is well organized, the memory subsystem has a performance close to the speed of the fastest memory, while costing only a little more than the amount of money required for the cheapest (and slowest) parts. The concept has been well known for a long time; Kilburn et al. [KELS62] proposed the idea of having an automatically managed two–level memory hierarchy in 1962 already.

To discuss the very basics of a memory subsystem is beyond the scope of this thesis; a well written technical introduction can be found in [PH98]. A general elaboration on how memory is managed by the operating system can be found in [Tan01]. A few important basics for this thesis are discussed in the following sections.

A modern memory hierarchy consists of at least 4 levels (more common are 5 or more), which are built on top of each other like a pyramid (see fig. 2.1). Some characteristics of these levels will be discussed in the following sections, with special emphasis on caches in section 2.1.1.

The single most important principle, which makes the use of memory hierarchies feasible, is called *locality*. This principle plays an important role for this thesis, which is why it is being explained in more detail here. It states that programs tend to use only a limited subset of their address space at any given point in time, or phrased differently: they do not jump around in it. The literature further isolates two kinds of locality:

1. *Temporal locality (locality in time): If an item is referenced, it will tend to be*

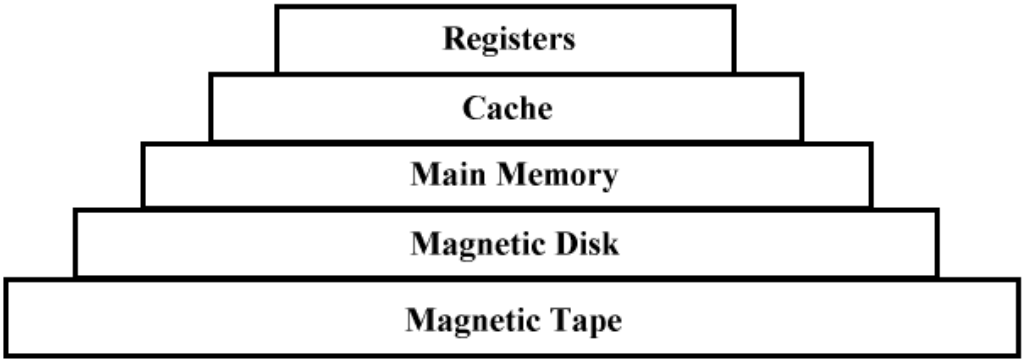


Figure 2.1: A memory hierarchy with 5 levels

referenced again soon.

2. *Spatial locality (locality in space): If an item is referenced, items whose addresses are close by will tend to be referenced soon.*

([PH98, p. 540])

[Wol96] additionally introduces a third kind, called *sequential locality* as a special case of spatial locality, where the two mentioned items have consecutive memory locations. Another common distinction is between *code locality* and *data locality*. The principle of locality can be derived from structural observations of programs, as well as from common data access patterns. The explicit reasons are outlined in table 2.1.

	temporal locality	**spatial locality**
code locality	programs spend a great amount of their time in loops, accessing the same lines of code over and over again	programs are processed sequentially
data locality	programs spend a great amount of their time in loops, accessing the same values in the process	data are often stored in arrays or structures, which are saved in memory together and are more often than not accessed sequentially

Table 2.1: Reasons for locality

Only for these reasons, the utilization of memory hierarchies is possible, since the few instructions and data items, which are in use at any given point in time, can be held in fast memory.

Spatial locality is also the reason, why every level in the memory hierarchy (except the topmost level) is divided into smaller units. These are usually called *blocks*. When an item is requested from a lower level of memory, the whole block is delivered and saved. Because of locality, there is a high probability that the next or a following access (may it be data or

instruction) will also request an item in the same block. If this happens, it is called a *hit*, because the requested item is already present in fast memory. A failure to find the requested entry is correspondingly called a *miss*. Misses will be further classified in section 2.1.1.

One of the most important measures for the performance of each level of the memory hierarchy is derived directly from the above definition: the so called *hit rate* (also: *hit ratio*). It is defined as the percentage of memory–accesses found in a higher (and therefore faster) memory level. The *miss rate* is correspondingly defined as the percentage of memory–accesses not found in a higher memory level, which equals $1-$ *hit rate*. Another term worth mentioning here is the so called *miss penalty*, which is the amount of time it takes to load a block from the lower level to a higher level in the hierarchy. These numbers are important, and therefore the performance of a level in the memory hierarchy boils down to the following formula:

$$MemoryStallTime = MemoryAccessNumber * MissRate * MissPenalty \quad (2.1)$$

These clock cycles need to be minimized for optimum performance, may it be by changing the algorithm, program or target hardware. The concepts described above are similar for all levels in the memory hierarchy, although the used terms may vary. For example, main memory is divided into *pages*, and a failure to locate an entry there is called a *page fault*.

Let us move on to a more complete tour through the memory hierarchy, starting with the top level, called *registers*. The registers are the fastest and most expensive part (per byte) of the memory subsystem. They are usually located right on the processor die. While there are only a few of them, their access time is usually high enough to fetch or write one or more register values per processor cycle. Registers are logically and sometimes even physically divided into general and special purpose registers. For general purpose registers, the program (and thereby the compiler or assembler programmer) has to decide, which values to put into them. They are usually used for important local variables and intermediate results of calculations. No hardware is involved in placing, replacing and identifying the proper data. Special purpose registers are heavily architecture dependent, some common ones include the *program counter* and the *stack pointer*, to name just a few. A more thourough description of registers and their position in the memory hierarchy can be found in [TG99].

The second level of any recent memory hierarchy is called the *cache*, and it is described in detail in section 2.1.1.

Main memory (also: *Random Access Memory* — RAM), the third level, is managed entirely by the operating system. Modern operating systems and computer architectures employ a memory organization called *virtual memory*. Tanenbaum describes virtual memory in the following way:

> *The basic idea behind virtual memory is that the combined size of the program, data, and stack may exceed the amount of physical memory available for it. The operating system keeps those parts of the program currently in use in main memory, and the rest on disk.* ([Tan01, p. 202])

There are several advantages to this approach that are outlined in detail in [PH98] and [Tan01], along with many other things to be said about memory hierarchies, virtual memory, main memory, magnetic disks and magnetic tapes, which are beyond the scope of this thesis. Our main focus of interest is caches, and they are treated in the next section.

2.1.1 Caches

The second level of any recent memory hierarchy is called *cache*. The literature defines the term as follows:

> *Cache is the name given to the first level of the memory hierarchy encountered once the address leaves the CPU. Since the principle of locality applies at many levels, and taking advantage of locality to improve performance is so popular, the term cache is now applied whenever buffering is employed to reuse commonly occurring items, examples include file caches, name caches, and so on.* ([HP03, p. 393])

For this thesis, we only use the word to describe the first meaning presented in the above quote. In our days, caches are usually divided into instruction cache and data cache for better performance and locality. This design is known as `Harvard Architecture`. Furthermore, it is common for computers to have two, or even three levels of caches, each of which is bigger and slower than the former. All of them are usually several times faster than main memory. The physical location of caches is strongly architecture dependent. The first level is typically placed right on the processor die, and often the second level is located there as well (as is the case for the `Pentium 4` architecture). The third level (if there is one) usually resides on the mainboard, connected to the processor through a high–speed bus.

The first important question to consider about the architecture of caches, should always be: Where can a certain block (blocks are also called *cache lines* at this level) reside in the cache? Hennessy and Patterson outline the three possible answers as follows:

1. *If each block has only one place it can appear in the cache, the cache is said to be direct mapped. The mapping is usually*

   (Block address) MOD (Number of blocks in cache)

2. *If a block can be placed anywhere in the cache, the cache is said to be fully associative.*

3. *If a block can be placed in a restricted set of places in the cache, the cache is said to be set associative. A set is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. The set is usually chosen by bit selection; that is,*

   (Block address) MOD (Number of sets in cache)

> *If there are n blocks in a set, the cache placement is called n–way set associative.*
>
> ([HP03, p. 397])

The above distinction may seem only marginally important, but there are huge gains to be achieved by changing cache placement to a more suitable mapping. Naturally, every strategy has its pros and cons. Direct mapped caches on the one hand only need little extra hardware to determine the position, where a block is to be placed, since there is only one possible position for the block. Fully associative caches on the other hand need more hardware, but are able to employ more advanced block replacement strategies (discussed below), which might be able to significantly reduce cache misses. Most modern architectures therefore settle for a compromise, which in this case leads to n–way set associative solutions.

Before further going into detail on caches, let us introduce a classification of *cache misses*. The literature describes three categories of these (commonly called the three Cs):

> 1. *Compulsory misses: These are cache misses caused by the first access to a block that has never been in the cache. These are also called cold–start misses.*
> 2. *Capacity misses: These are cache misses caused when the cache cannot contain all the blocks needed during execution of a program. Capacity misses occur because of blocks being replaced and later retrieved when accessed.*
> 3. *Conflict misses: These are cache misses that occur in set–associative or direct–mapped caches when multiple blocks compete for the same set. Conflict misses are those misses in a direct–mapped or set–associative cache that are eliminated in a fully associative cache of the same size. These cache misses are also called collision misses.*
>
> ([PH98, p. 609])

Various approaches can be taken to decrease the misses. Compulsory misses can only be reduced by increasing the block size, requiring more memory bandwidth in return and possibly also increasing the miss penalty. Capacity misses have the potential to be reduced by enlarging memory (in our case the cache), which has been done for all architectures at all times, because all kinds of memory tend to get cheaper and cheaper. Conflict misses can be reduced by architectural rearrangements, some of them are described later in this section.

The placement of blocks directly leads to the next problem: How does one identify, whether or not a wanted block is in the cache? The natural solution is the employment of address tags for each position. These store the virtual memory address that their cache block currently holds. In direct mapped caches, only one tag needs to be checked per date access, for associative solutions more than one position must be looked at. Doing this sequentially would greatly

reduce performance, so even more extra hardware is needed to check each possible position in parallel.

Furthermore, there needs to be a way to know, whether or not the entry is identical to the one presently stored at the memory location the address tag points to (this might not be the case after context switches, or in multiprocessor systems where more than one processor can access a memory location). Each cache position therefore has a so–called *valid bit* attached, which has this information stored.

A question subject to much research in the past is: Which block should be replaced in case of a cache miss? For direct mapped caches this is an easy choice, since only one entry is a candidate for replacement. For associative caches there are a lot of possible algorithms to be considered, among them *random replacement* or *least–recently used*, which are discussed in a variety of publications, e. g. [PH98].

Although a great majority of cache accesses are reads, the write strategy for caches has a considerable impact on the performance of the memory subsystem as a whole. Two possible strategies are commonly deployed: *write–through* basically means that any information is written to the lower level memory and the cache simultaneously. A heavy performance penalty is to be paid here, since the processor needs to stall until the lower level memory has finished writing the block. One or more write buffers can help reduce this time frame, and are often used. The advantages of this strategy include ease of implementation and the assurance that the lower level memory always has the most recent data, which is important especially for parallel processors. Most modern architectures use a different cache write strategy, which is called *write–back*. On the one hand, information is only written to the block in the cache, making writes a fast operation on these machines. Especially when data are changing rapidly while being held in the cache, many writes to the lower memory level can be avoided. On the other hand, blocks must be written out to this level when they are being replaced, which may result in processor stalls later (that may slow down read operations). An important performance optimization is the so called *dirty bit*, which indicates whether or not the block has been written to. If the bit is not set and the block is to be replaced, it needs not be written back to main memory, saving important memory bus bandwidth. Another disadvantage of this strategy is the fact that the lower level memory does not always have the current data, which leads to a whole new set of problems for parallel processors, which are discussed in section 2.2.3.

There is yet another problem to consider when doing data writes: What happens, if the entry to be written is not in the cache already — i. e. what happens on a write miss? Two options are considered here, the first being *write–allocate*. The block in question is loaded into the cache first for this strategy, and then modified as usual. The second possible solution is correspondingly called *no–write–allocate*.The block in question is directly modified at its location in the lower level memory then. The *no–write–allocate* strategy is usually coupled with a write–through cache write solution, while write–allocate usually gets deployed along with write–back, although theoretically other combinations are possible.

2.2 An Overview of Parallelism

The need for parallelization has already been emphasized in the introductory chapter. In general, there are two main goals to be pursued when talking about parallelism. The first one is high availability, which is sought especially in mission critical systems, including various kinds of servers (e. g. web servers, database servers, application servers etc.). These systems use redundant parts (e. g. multiple processors, subsystems or even whole computers) to continue operation even in case of one or multiple hardware failures, since defective parts can just be swapped out without interrupting the operation of the system as a whole (and all this is available at a highly competitive price, since most of the time off–the–shelf components can be used not only for the processors, but also for IO- and memory subsystems). Parallel machines built for this purpose are not covered here, since we focus on the second goal to be pursued by parallel architectures: performance. Patterson and Hennessy summarized this goal in a neat way, when they wrote:

> *Computer architects have always sought the El Dorado of computer design: to create powerful computers simply by connecting many existing smaller ones. This golden vision is the fountainhead of multiprocessors.* ([PH98, p. 712])

Unfortunately, it is not quite that easy, as section 2.2.1 will outline. A more special treatment of the concepts involved in parallelism follows in the next subsections.

2.2.1 Speedup, Efficiency and Amdahl's Law

How much is to be gained by using parallel machines? How do multiprocessors affect performance? Unfortunately, even for uniprocessor machines there is no consensus about how to measure performance at all. Probably the most universally accepted measure remains the simplest one: *time*, which is often more accurately denoted by synonyms like *wall–clock time*, *response time* or *elapsed time*. They correspond to the time difference between the beginning and the end of a computational activity.

Today multitasking machines are in common use, therefore *CPU execution time* is often a more meaningful measure, since it only measures the time the CPU spends on the program in question. There is a problem with using *CPU execution time* for parallel problems though, because it does not include the time the processors spend waiting for communication or synchronization. Thus, fundamental problems with the parallel algorithm will not show up, and this is why we utilize *wall–clock time* in this thesis. The problem with multitasking mentioned above is partially dealt with by running all tests multiple times, using only the best numbers measured. It would have been even better to run the experiments on an otherwise unloaded machine, but this was not possible for organizational reasons.

Other measures of performance include *throughput* and *latency*. These measures are useful to evaluate different architectures, computers or operating systems, but not so interesting for

individual programs, which is the reason why they are not covered here. Quite often various forms of *benchmarks* (such as the *SPEC processor benchmark suite*) are used for comparison. With benchmarks, the execution time of an agreed upon set of sample programs is measured, which allows for comparisons of different architectures, compilers etc. Benchmarks are beyond the scope of this work, and are covered in detail in [PH98] and [RR00].

The most widely used performance measure for parallel machines is called *speedup*. Speedups are used and defined in a variety of ways, all of which boil down to a formula like:

$$Speedup(n) = T(1)/T(n) \tag{2.2}$$

The speedup to be achieved with n processors is therefore defined as the time it takes to solve a problem on a single processor, divided by the time required on n processors. While this formula seems relatively straightforward, variations exist in how the times are calculated and by how much the algorithm is allowed to be changed. For example, a common trick to demonstrate high speedups is to compare the time reached by the parallel algorithm to the time achieved running the same code on a single processor, and not to a serial version of the same algorithm. In this way, synchronization and communication efforts in the parallel version are being underrated. For this thesis, the single processor algorithm as well as the multi processor one are allowed to be optimized as much as possible, which leads to smaller speedups, since the communication and synchronization overhead required for parallel programming is allowed to be skipped in the single processor case. A typical graphical representation of a speedup curve is shown in figure 2.2, adapted from [BS96].

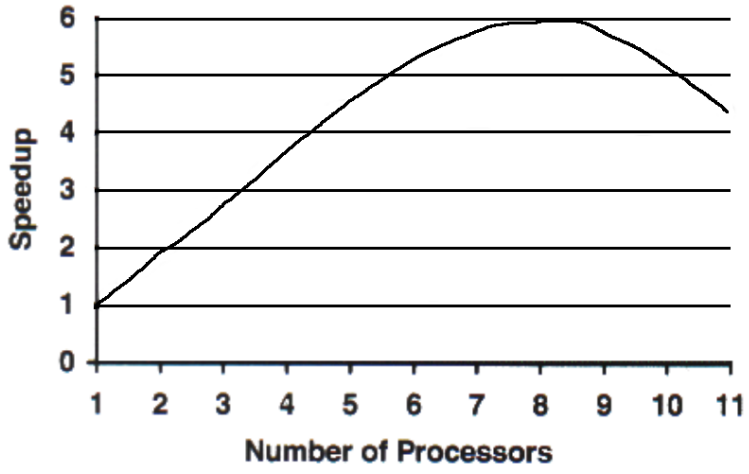


Figure 2.2: Sample speedup representation

A common phenomenon can be observed here: The speedup curve is not monotonously rising, but reaches a certain saturation point (at 8 processors in figure 2.2). Above that, the additional processing speed gained by adding more processors is consumed by interprocessor communication and synchronization, as well as by expenses paid for process- / thread–management and insufficient load–balancing. While this saturation point is different for any given parallel problem, there will always be such a point. In 1967 Gene Amdahl formally presented his stab at

the problem in [Amd67], by introducing a formalism now know as *Amdahl's Law*. He divides every problem into a parallelizable fraction p, and a serial fraction $s = 1 - p$. If inserted into the formula for speedup stated above (with n = number of processors), we get:

$$Speedup(n) = 1/(s + p/n) \tag{2.3}$$

Note that even with an infinite amount of processors, the speedup can not exceed $1/s$, which is exactly the behavior observed in the example presented above.

There is an interesting exception from the assumptions stated. Sometimes, speedups higher than the number of processors can be achieved. How is this possible? These so called *superlinear speedups* are related to the larger memory subsystems used, when many processors come into play. Often, n processors have n times the caches and memories available, therefore eliminating I/O–bottlenecks in the process, making program execution faster, because e. g. all the needed data items fit into the caches. This may cause great improvements to performance.

There is another measure for performance in parallel processing called *parallel efficiency*. It is derived from the speedup mentioned above, and calculated using the formula:

$$Efficiency(n) = Speedup(n)/n \tag{2.4}$$

This measure indicates how much speedup is gained per additional processor. Therefore, it is an alternative indication, how well a parallel problem (or algorithm or implementation) is suited for parallelization. A typical efficiency curve, derived from the speedup example above, is shown in figure 2.3, also adapted from [BS96].

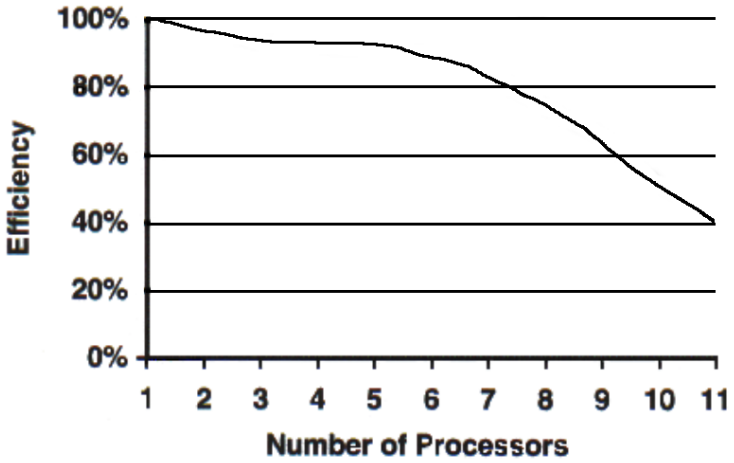


Figure 2.3: Sample efficiency representation

For a more thorough analysis of performance measures of parallel systems consult [RR00].

2.2.2 Flynn's Classification

One of the first and most famous classifications of parallel architectures was introduced by Flynn in 1966 [Fly66]. Flynn distinguishes between the instruction and data stream, which

leads to the following classes of computer architectures: SISD, SIMD, MISD and MIMD (also shown in figure 2.4, the important architectures for this thesis are highlighted). Of course, this categorization is only a coarse one, since there are some hybrids.

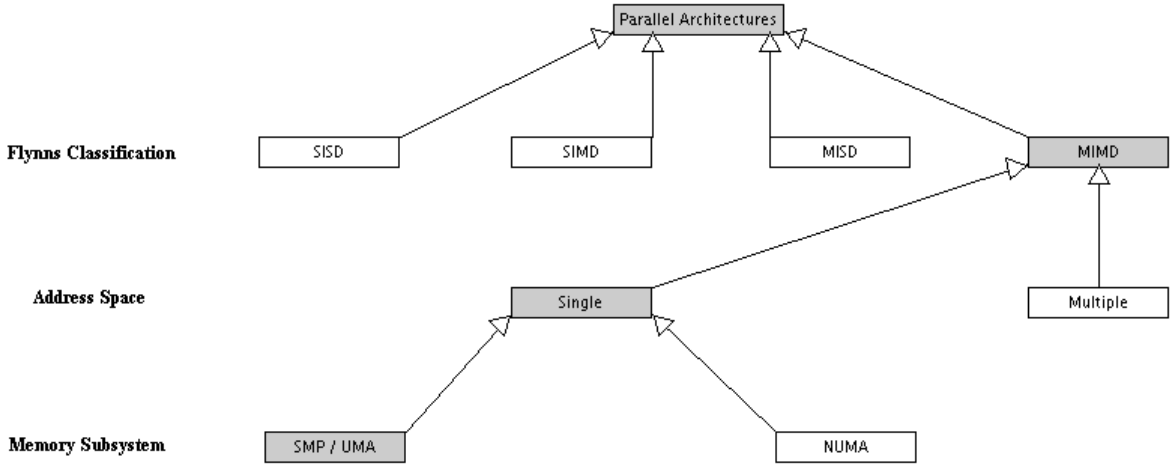


Figure 2.4: A classification of parallel architectures

SISD is an abbreviation for *Single Instruction Single Data*, which basically translates into one instruction working on one data item at a time. All sequential, non–parallel computers fall into this category, it is therefore not interesting for the scope of this thesis.

SIMD (*Single Instruction Multiple Data*) machines are able to apply one instruction to multiple data in each clock cycle. Many older computer architectures have been built this way, e. g. the Connection Machine (CM-1 and CM-2), as well as the Goodyear Massively Parallel Processor. These usually had their nodes aligned in a grid–like layout, connected by a single or multiple buses. The processors had the normal arithmetic capabilities of their time, but lacked a program control unit, making them simpler and cheaper. This was one of the reasons that the CM-2 could be equipped with up to 65000 nodes. The program control unit was laid out separately, broadcasting the same instruction to all processors, which was then processed simultaneously, but using different data sets. For more flexibility, each unit could be turned off separately, allowing for *if*–clauses. The reason why these machines became popular at their time was their superior ability to deal with large arrays. Many challenging problems involve similar operations on huge sets of arrays, which could be dealt with efficiently and without the overhead of a memory–access and instruction decoding for every single processor. In our days however with ever decreasing hardware costs, more flexibility was necessary, and therefore traditional SIMD machines are hardly in use anymore. The SIMD concept has been reused in different ways though, for example in recent enhancements to the x86 instruction set, called MMX or ISSE. Vector computers also belong into this category, but are not further discussed here.

MISD or *Multiple Instruction Single Data* is the third group in Flynn's classification. There is no machine in practical use today to fit this description, although some pipelined architectures have similarities to this group.

The last, and most important class of parallel computer architectures today is MIMD, which stands for *Multiple Instruction Multiple Data*. Here every processor operates on its own stream of data and instructions. This is the most expensive approach from a hardware designers point of view, since every unit needs to fetch instructions as well as data at each and every clock cycle, requiring efficient memory–access for each processor. Therefore section 2.2.3 deals with different solutions to memory organization for parallel architectures. The mentioned disadvantage is more than compensated by two large advantages. The first is the ability to use off–the–shelf microprocessors, which makes this approach even cheaper than SIMD, from a customers point of view. The big parallel machines of our times use the same processors as small 2–4 processor servers, which in turn use only slightly modified versions (usually with a bigger cache) of the ones used in workstations. Economy of scale is all that is needed to make this approach worth pursuing. The second advantage is added flexibility. From using the MIMD–machine as a single user solution for one highly demanding application, to letting thousands of processes and/or users work on it at the same time, everything is possible and only a matter of tuning some operating system parameters. For the reasons stated, MIMD has risen as the architecture of choice for todays supercomputing demands, and for the rest of this work we are going to concentrate on it.

2.2.3 Memory Organization in Parallel Architectures

Existing MIMD architectures can be coarsely divided into two groups by the layout of their memory subsystems. There are systems with a single address space, and systems with multiple address spaces. Each of these classes have subgroups, the relevant ones are shown in figure 2.4. The next two sections are covering them in more detail.

Single Address Space Architectures

In Single Address Space architectures, all processors share the same address space, and therefore each word in memory is accessible for all units using the same address. Of course, this makes life for programmers and compiler writers relatively easy, while hardware designers may have to jump through some additional hoops to make it possible. The second main advantage of Single Address Space machines is their memory efficiency, since every program (including the operating system) only needs to be stored once. There are two important subgroups for this class of systems, which are explained in more detail in the rest of this section.

The first one is called Symmetric Multiprocessors (SMP) or Uniform Memory Access (UMA) architectures. These computers deploy a configuration, in which each processor is able to access any word in memory using the same amount of time, independently from where the word is stored (see fig. 2.5, adapted from [HPC]). This is physically accomplished by using either fast buses, crossbars or networked structures. SMPs are popular for small servers, since their architecture is relatively simple. Examples of this class are often based on Intel's Xeon

or AMD's Opteron processors in our days. Scaling this approach to tens or even hundreds of nodes is increasingly difficult though, since the size of the interconnection network between the processors and memory banks increases at least linearly and usually much worse with the number of units to connect.

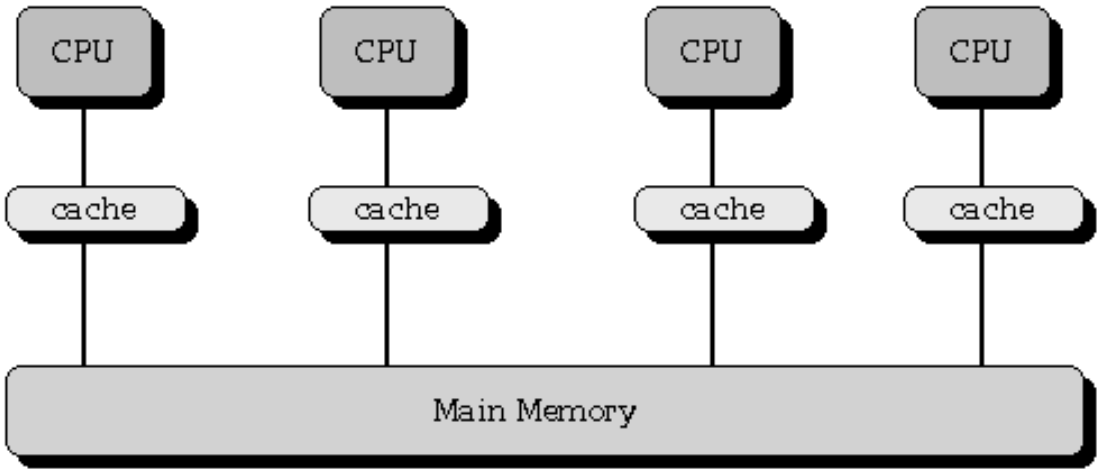


Figure 2.5: A Uniform Memory Access architecture

For large numbers of processors the second group, called Nonuniform Memory Access (NUMA) architectures, is the dominant one. There are two levels of memory involved here, one of which is called local, since access to it is relatively fast for a certain processor (because this part of memory is located physically close), while the other level (which is usually much bigger) is called remote memory. That kind is located close to other processors, but can still be accessed with a speed penalty (see figure 2.6, also adapted from [HPC]). Besides the ability to easily scale this approach to hundreds or thousands of processors without overstressing the interconnecting network, one more advantage of this architecture is not to be left unmentioned: Accesses to the local memory can be made fast, so when most of the data one processor needs can be held locally, NUMAs are able to outperform UMA architectures. The key disadvantage of NUMA architectures is expensive interprocessor communication, which is covered in detail in section 2.2.4. How much of an impact this disadvantage leaves, depends (as always) on the application, and therefore no general recommendations for or against certain architectures should be made without considering the target application first.

There are a couple of hybrid approaches to memory architecture, which aim at finding a suitable compromise between the two extremes. The SPP2000 for example is equipped with 48 processors. Each 16 of these constitute a hypernode, which can be considered an SMP computer, while all of them together form a NUMA. More flexibility is the result.

Multiple Address Spaces Architectures

The common characteristic of Multiple Address Spaces architectures is, once again, their unique memory organization. Each processor has its own memory attached to it, which

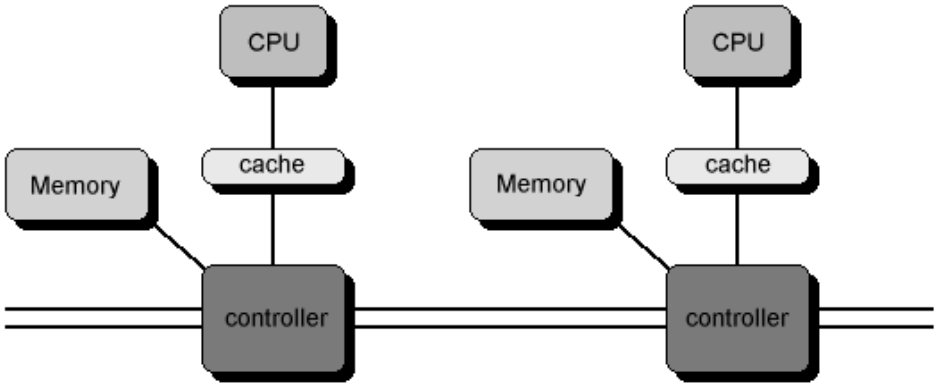


Figure 2.6: A Nonuniform Memory Access architecture

is accessible only to itself. This allows for quite simple designs, as far as hardware is concerned, and also allows these approaches to easily scale to huge amounts of nodes (machines with several thousand units are possible, and already commissioned). Furthermore, the use of caches is relatively straightforward in `Multiple Address Spaces` architectures, since every processor has its own set of memory and caches (more on the problems of using caches in multiprocessor systems is to be found in section 2.2.3).

There are some disadvantages to be mentioned though. First, there are high costs of administration, usually the costs for administrating a multiple address space system with n processors is equal to the costs associated with administrating n single address space machines. Second, these machines are more difficult for programmers to use efficiently, because shared memory interprocessor communication is usually not available (more on this topic in section 2.2.4). Third, all applications need to be stored in memory n times, including the operating system, making these systems relatively memory–inefficient.

In some literature, `Multiple Address Spaces` machines are divided into two subgroups, the first one being `NORMA` or `No Remote Memory Access` machines and the second one being `Clusters`. They are basically separated by their level of integration, since in `NORMA`s the processing nodes are usually physically close together and connected by high–speed networks over short distances, while `Clusters` may be spread across whole buildings, campuses or companies. This distinction is quite fuzzy though, and while the interconnecting networks of clusters are gaining speed by the minute, the term `NORMA` is slowly becoming obsolete.

There is one relatively new architectural design in `Multiple Address Spaces` architectures, called `Grids`. Again, the distinction to `Clusters` is not entirely clear, the main point possibly being size and heterogeneity. A `Grid` is usually not assembled from workstations or PCs, but is built by connecting whole `Clusters` and/or high performance parallel machines (e. g. `SIMD`s). Furthermore, the physical distances between the nodes can be even larger than in `Clusters`, in some cases even as far as worldwide. Often, TCP/IP network connections are used for communication between the units. Of course, these have high laten-

cies and communication costs involved, limiting the general use of Grids somewhat. For more information on Clusters or Grids consult [Leo01a], since they are not in the main focus of this thesis.

Caches in Multiprocessor Systems

There are a number of pitfalls to be considered when dealing with caches on parallel machines. *Cache coherence*, for example, is a huge issue for Single Address Space architectures, while it is basically not a problem at all for Multiple Address Spaces ones (a big advantage of these). Therefore, in this chapter only the first ones are considered.

But let us start with a short problem description. In any reasonably modern parallel system, each processor accesses memory through its own caches. If the needed data item is found in the corresponding cache, no memory–access is taking place. But, if another processor had changed the value while the cache was buffering it, the first processor would have no knowledge of this fact, and would therefore use an old and possibly incorrect value. The picture gets even worse, when write–back caches (introduced in section 2.1.1) are used, since the current right value might not even be stored in main memory at the time. Solutions to this dilemma are summarized under the term *cache coherence*, a more exact definition (in this case of the more general term *coherence*) would be:

> *A memory system is coherent if*
>
> 1. *A read by a processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.*
> 2. *A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.*
> 3. *Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.*
>
> ([HP03, p. 550])

There are two main approaches to cache coherence for NUMA architectures, the first one is called non-cache coherent NUMA (nccNUMA) and it basically lacks any hardware support for the issues described above. On the one hand, only data from local memory are cached, which of course introduces a big performance loss for these machines. On the other hand, less expensive and more easily scalable hardware can be designed this way. An example is the Cray T3E.

The second approach is correspondingly called cache coherent NUMA (ccNUMA), and it has built–in support for cache coherence, making these machines faster, but also more expensive and less scalable. An example is the Sequent NUMA-Q 2000.

For SMP architectures, there is no non–cache coherent case, since the concept of local memory does not apply. Therefore, every SMP system needs some kind of hardware support for cache coherence.

Two main protocols are used by hardware manufacturers to guarantee cache coherence. Their main purpose is to keep track of any sharing of data blocks among different processors. The first one is called *directory based*. It implies that the sharing status of every block of physical memory is kept in a single location — the so called *directory*, where any processor can look up the status prior to reading or writing. Directories are heavily used on ccNUMA architectures. This thesis concentrates on SMPs, therefore a detailed description of the protocol is omitted here and can be found e. g. in [HP03].

The second approach is called *snooping*. The sharing status of any physical location is kept together with the corresponding block in every cache the block resides in, thereby saving the centralized state tables. This protocol is usually implemented on a shared–memory bus, where all cache controllers can *snoop* on the bus to determine, whether or not they have a copy of the block that is being requested, and can act if necessary. Possible reactions include:

1. *Write–invalidate: The writing processor causes all copies in the other caches to be invalidated before changing its local copy; it is then free to update the local data until another processor asks for it. The writing processor issues an invalidation signal over the bus, and all caches check to see if they have a copy; if so, they must invalidate the block containing the word. Thus, this scheme allows multiple readers but only a single writer.*
2. *Write–update: Rather than invalidate every block that is shared, the writing processor broadcasts the new data over the bus; all copies are then updated with the new value. This scheme, also called write–broadcast, continuously broadcasts writes to shared data, while write–invalidate deletes all other copies so that there is only one local copy for subsequent writes.*

([PH98, p. 721])

Write–update is quite often used together with write–through, because all writes have to travel across the memory bus anyways. Write–invalidate on the other hand only causes bus activity once, thereby saving precious memory bandwidth. Write–back behaves similarly, and since for most modern multiprocessors the shared memory bus is a bottleneck, write–back and write–invalidate (or slight variations of these, e. g. the *MESI* protocol used in the Pentium and PowerPC architectures) are the protocols of choice for most commercial parallel machines.

The implementation details of these approaches are beyond the scope of this thesis, and are discussed in great detail in [HP03].

2.2.4 Interprocessor Communication

One of the most important decisions when programming on a parallel architecture is, how the communication between the nodes is to be accomplished logically. Two main concepts have emerged in the past, which are derived from the memory organizations described in 2.2.3. First, there is the *shared memory communication* approach. Data are exchanged between processors by storing them in a shared memory area, where they can be retrieved by the communication partner with a simple load. Some kind of shared memory is needed for this approach to work, and thats why it is usually used on SMP and NUMA machines. Advantages include low overhead and ease of programming. Implementations of the shared memory concept include various kinds of *threads* (e. g. pthreads, Java threads), several directives for parallelism (e. g. Linda or OpenMP, covered in chapter 2.2.5) and of course some proprietary vendor solutions.

The second approach to interprocessor communication is called *message passing communication*. Data are exchanged by passing explicit messages in–between the communication partners there, including the values to be transmitted. No common address space is needed, merely some form of identification for each processor must be known. The main advantage of message passing is the lack of requirements for certain hardware, since almost every parallel architecture natively supports some form of message passing. Another advantage, mentioned in [HP03], is the requirement for compiler writers and programmers to pay close attention to communication, which might result in efficiently designed solutions. A couple of standards have been proposed and implemented, the most well known of these are probably the Message Passing Interface (MPI) and the Parallel Virtual Machine (PVM). More information on message passing communication in general can be found in [WA99] and [RR00], a complete reference to MPI is [GNL98], and information about PVM can be found in [Gei94].

As already mentioned, the shared memory approach is closely coupled with Single Address Space architectures (especially SMPs), while message passing is usually deployed across Multiple Address Spaces machines. This has historical and architectural reasons, but since both variants for interprocessor communication are merely models, message passing is possible on e. g. SMPs, too (and actually quite easy to implement by copying memory locations), as well as shared memory approaches on Clusters (e. g. by providing shared virtual memories, usually with a special software layer, called *middleware*). A more thorough discussion of interprocessor communication can be found in [Leo01a] and [RR00].

2.2.5 An Introduction to OpenMP

OpenMP is an Application Programming Interface (API), which represents a standard for shared memory communication. It is heavily used for the example programs in chapter 5, and therefore a short introduction to OpenMP is presented here. For further coverage of this topic consult one of the various resources on the internet or in books, including the nice overview in [Lab03] or a more extensive coverage in [CDK00].

Advantages of OpenMP include *portability* (it runs on a variety of architectures, including most UNIX platforms and various kinds of Windows operating systems and is specified for C/C++ and Fortran), *standardization* (may even become an ANSI standard later), *simplicity* (it employs only a limited set of directives) and *ease of use* (it allows incremental parallelization of serial programs, making the first steps to parallelism relatively easy).

OpenMP internally uses threads for parallelism. These must be explicitly requested by the programmer with compiler directives. The number of threads to create can be adjusted dynamically, even by the runtime environment. Nested parallelism is also available, meaning it is possible for threaded regions to create even more threads.

In contrast to simple thread–based programming models, OpenMP imposes a structure on all tasks. The program starts its life with a single thread, the so called *master thread*. This thread may then start a *parallel region*, and distribute its work across several *children*. A sample program execution scheme is presented in figure 2.7 ([Lab03]), a short piece of C code, showing how to explicitly request a number of threads, adapted from [Lab03], is presented in figure 2.8.

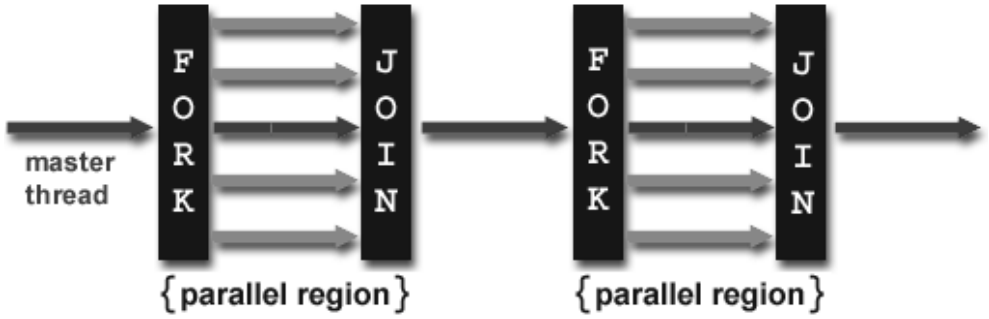


Figure 2.7: Program execution scheme for OpenMP programs

Besides explicit work distribution as in figure 2.8, there are three kinds of basic work sharing constructs in OpenMP, which divide the execution of the enclosed code region among different threads (that must have been created beforehand, using the parallel construct described above). The first one of these is the *do / for* directive. The iterations of the loop directly following the directive are executed in parallel. How the work is divided among the different threads is configurable by using the *schedule* clause. A non–iterative work sharing construct is made available through the *sections* directive. Here the enclosed sections of code are divided among the threads in the team, so that each section is carried out exactly once. The last work sharing construct is called *single*. Every piece of code enclosed by this clause is executed by only a single thread, making this useful for code, which is not thread–safe. All three kinds of work sharing constructs are visually presented in figure 2.9, adapted from [Lab03].

Besides sharing work, the single most important topic to consider about parallel regions is, how to synchronize the threads. Several constructs for this purpose are available in OpenMP, presented here in no particular order:

1. *critical*: specifies a region, which is to be executed by only one thread at a time (all others

```
#include <omp.h>
int main () {
    int nthreads, tid;
    /* Fork a team of threads giving them their own copies of variables */
    #pragma omp parallel private(nthreads, tid) {

        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    } /* All threads join master thread and terminate */
}
```

Figure 2.8: A sample C program, including OpenMP directives

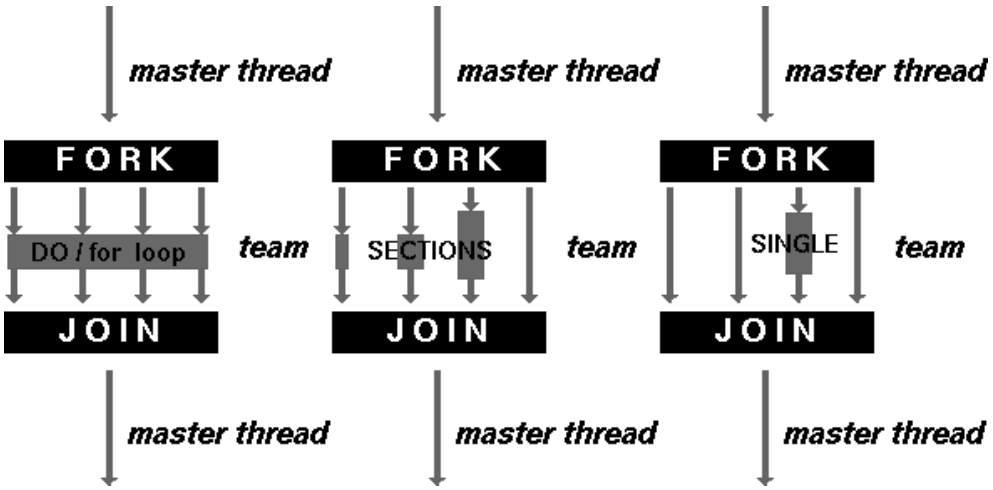


Figure 2.9: Work sharing constructs in OpenMP

block until the the first thread exits the region)

2. *barrier*: specifies that all threads must wait at this point, until all other threads have reached the barrier
3. *atomic*: specifies that a certain memory location must be updated atomically (meaning that only one thread is allowed to write to this location at a time)
4. *flush*: specifies a synchronization point, at which all thread–shared variables must be written back to memory (thereby providing a consistent view of memory for all threads)
5. *ordered*: specifies that for some code within a loop, the iterations will be executed in order (meaning, in the same way as if the loop was run on a serial processor)

`OpenMP` can neither provide facilities for distributed memory programming, nor is it necessarily implemented identically across different architectures or by different vendors. Furthermore, it makes no promises about performance whatsoever. Nevertheless, its ability to generate portable source code compensates for all of these disadvantages, and makes it the best choice for parallel development in shared address space environments at the time of this writing.

2.2.6 Automatic Parallelization and Dependencies

Writing parallel code remains a burden on the programmer, which would best be delegated to automated tools. Some of these tools exist, usually implemented as *parallelizing compilers*, and their objectives are summarized as follows:

> *Automatic parallelization transforms an imperative sequential input program into a parallel program automatically. At present, chiefly data parallelism is exploited. It is introduced by restructuring and parallelizing the sequential loops.*
>
> ([Leo01a, p. 178])

Some different aspects are emphasized by this definition:

> *Automatic parallelization techniques aim at extracting the parallelism out of a source program. The extraction is automatic, i. e. by the compiler, and is architecture–independent.* ([DRV00, p. xiii])

Several possible candidate languages for automatic parallelization exist, the most common ones in our days are `Fortran`, `C`, `C++` and `Java`. At the time of this writing, these compilers do not aim at automatically parallelizing whole programs, but merely use a few well–known techniques to achieve parallelism for a limited subset of any given program. Above all, they concentrate on loops, which is reasonable since programs usually spend most of their time processing these. Especially *for*–loops seem to be a promising target.

To cover all the problems involved in automatic parallelization is way beyond the scope of this thesis, a brief introduction is given in [Leo01a]. There have also been several publications about improved compiler algorithms for automatic parallelization during the last few years, e. g. [LL98], [KS95], [KRC97] or [SM97], to name just a few.

However, there is a problematic area worth mentioning here, which plays a major role in the `iblOpt` programming tool, introduced in chapter 2.3 as well — *dependency analysis*.

In one of his publications, Wolfe defines three types of dependencies:

1. *Flow dependence occurs when a variable is assigned or defined in one statement and used in a subsequently executed statement.*
2. *Anti–dependence occurs when a variable is used in one statement and reassigned in a subsequently executed statement.*
3. *Output dependence occurs when a variable is assigned in one statement and reassigned in a subsequently executed statement.*

([Wol96, p. 137])

Wolfe also introduces graphical and textual representations of the different kinds of dependencies, which will be used throughout this thesis (and especially in the example programs of chapter 5). They are presented in table 2.2.

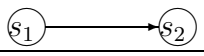
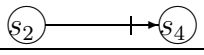
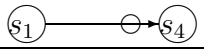
Dependency	Textual Representation	Graphical Representation
Flow	$s_1 \delta^f s_2$	
Anti	$s_2 \delta^a s_4$	
Output	$s_1 \delta^o s_4$	

Table 2.2: Representations of dependencies

Dependencies are so important, because they limit the potential for parallelization. All parallelizing compilers, as well as the `iblOpt` programming tool, need to honor them, because failure to do so would lead to incorrect code.

2.3 The Semi–Automatic Method and the `iblOpt` Programming Tool

This section describes the *semi–automatic method* and its main component, the `iblOpt` programming tool. The goal of the semi–automatic method is to provide an intermediate alternative between expensive locality–optimized code built by specialist programmers, and the cheap, but relatively unoptimized (for locality) code created by compilers.

The method was first introduced by Leopold in [Leo98], and is best described like this:

> *One human approach to locality optimization considers several small program instances of a given program, finds optimal or close to optimal statement sequences and data assignments for the instances, and generalizes their structure to the program. We suggest using this approach in a semi–automatic locality–optimization method. Roughly, the manual part comprises the specification of the program instances and the generalization of the results; the automatic part comprises the optimization of the program instances.* ([Leo98, p. 295])

A more verbose description of this process follows:

1. identify performance–critical parts of the program
2. transform them into "toy versions", by fixing parameters at small values in the process (*instantiation phase*)
3. let `iblOpt` find optimum statement and data orderings for the toy problem (*optimization phase*)
4. generalize the solution to the original program (*generalization phase*)

A graphical representation of this method is given in figure 2.10. In there, all phases are outlined with lighter colors, and N is used to indicate the size of the input parameters.

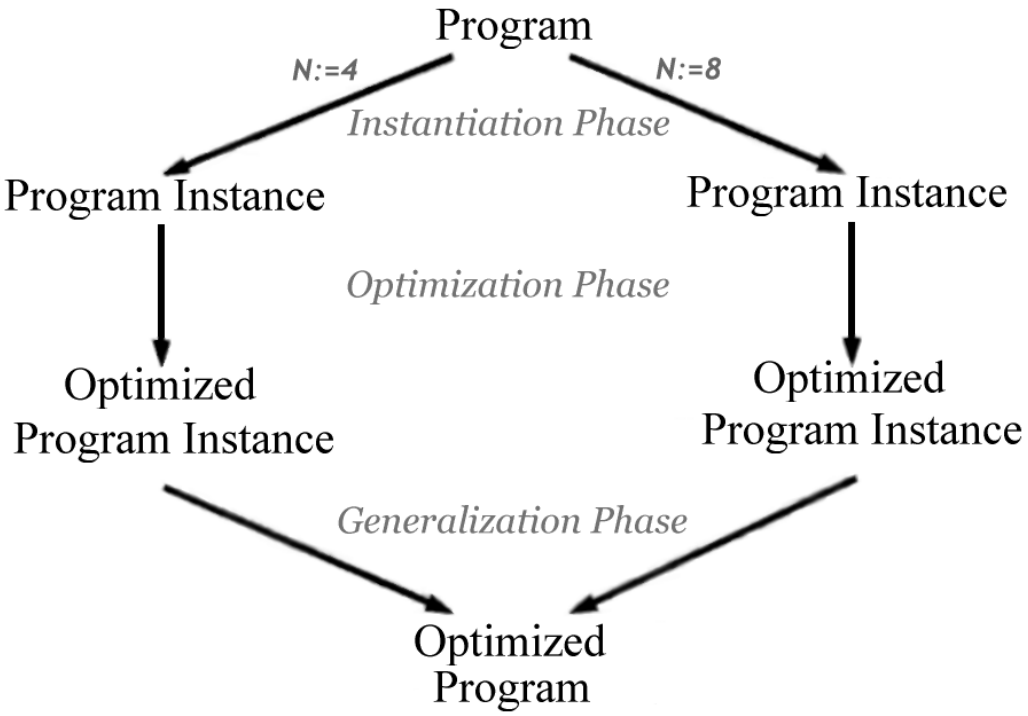


Figure 2.10: A graphical representation of the semi–automatic method

The `iblOpt` program is invoked during the optimization phase only; here is a more detailed description of this part:

> *The tool deploys a new method called instance–based locality optimization: After analyzing the input program, the user is asked to specify concrete values for loop bounds and other variables. The values must be small such that the program instance models the real program. By unrolling loops, the program instance is rewritten as a sequence of statement instances, which are then reordered for locality, using a local search algorithm with a specific objective function. Finally, the optimized instance is represented as a loop program, and the order of accesses to the program arrays is visualized.* ([Leo01c, p. 2])

In the above quote, two terms central to the described approach are used, namely *program instance* and *statement instance*. A short definition, which also includes the concept of a *data instance* follows:

> *The central concept of the semi–automatic method is that of a program instance (PI). A PI comprises a sequence of statement instances (SIs) that represent elementary or complex computational units. Each SI operates on a sequence of instantiated data elements (DI). A DI is a non–variable description of a location in memory.* ([Leo98, p. 2])

Implementation details of the algorithms used in `iblOpt` can be found in [Leo98]. An additional feature of the program is *regularity optimization*. It helps the programmer to identify regular patterns in the SI output of `iblOpt`, making it possible to more easily identify structures, e. g. loops in the generalization phase. A short definition of this capability would be:

> *In our context, regularity optimization is the task of rearranging the SIs of a PI so that the indices in the names of the SIs follow some regular patterns, as much as possible. Regularity optimization is successful if the resulting SI sequence can be expressed in a structured representation that is compact and intuitively appealing.* ([Leo01b, p. 428])

More details are omitted here (and can be found in [Leo01b]), because regularity optimization is not further emphasized in this thesis. A complete description of the `iblOpt` programming tool can be found in [Leo01c].

Chapter 3

Conceptual Work

The main contribution of this thesis is an algorithm, based on a *local search* strategy that has the ability to distribute a program instance (consisting of several statement instances) onto a predefined number of processors. *Local search* is a well–known algorithm for solving combinatorial optimization problems, as described in [AL97]. It works incrementally, taking one feasible (but possibly not good) solution as input and improving it stepwise.

Let us start by defining some terms. Let F be the set of feasible solutions to a specific problem, which is discriminated from the set of all solutions by a number of problem specific constraints. Therefore, $f \in F$ is a possible solution to the problem. For every $f \in F$ a set of *neighbors* $NB(f) \subseteq F$ can be defined, which is also called a *neighborhood*. Each $f_k \in NB(f)$ is similar to f in some problem specific way.

A generic local search algorithm takes one solution $f \in F$ as its input, and evaluates every $f_1, \ldots, f_n \in NB(f)$ using a problem specific *objective function* $o(f)$. The best $f_k \in NB(f)$ (concerning the objective function) then becomes f, and the algorithm starts anew. This process is repeated, until an optimum (in our case a minimum, optimizing for maximums is just as well possible, depending on the used objective function) is found, where:

$$\forall f_k \in NB(f) : o(f_k) \geq o(f) \tag{3.1}$$

When given enough time, local search algorithms always find optima, unfortunately quite often these are only local ones. One possible improvement (also used for this thesis) is repeating the local search with varying initial $f \in F$ for algorithm initialization, until a reasonably good solution is found. A pseudo–code implementation of the combined algorithms is given in figure 3.1.

Next, section 3.1 adapts the general local search approach to the needs of our specific problem domain.

repeat
  f := *initial random feasible solution*;
  $f_{best} := f$;
  repeat
    $f := f_{best}$;
    foreach $f_k \in NB(f)$
      if $o(f_k) < o(f_{best})$ **then** $f_{best} := f_k$;
    endforeach;
  until $o(f) <= o(f_{best})$;
until *reasonably good solution* $f \in F$ *is found*;

Figure 3.1: A generic local search algorithm, executed repeatedly to increase the chances of finding global optima

3.1 The Optimization Algorithm

Quite a few gaps need to be filled in order to adapt the general local search algorithm from section 3 to our needs. Before we look into these, let us state some formal assumptions:

- A *program instance* (*PI*) consists of several ordered *statement instances* (*SIs*) (see section 2.3 for definitions).
- Let us denote the SIs by $s_1, \ldots, s_n$, and let $S = \{s_1, \ldots, s_n\}$.
- Let us denote the DIs by $d_1, \ldots, d_n$, and let $D = \{d_1, \ldots, d_n\}$.
- Let n_p be the number of processors to optimize for.
- Let $|S|$ be the number of statement instances in a program instance.
- Let $|D|$ be the number of data instances in a program instance.
- Let D^* denote the set of all sequences (of any length ≥ 0) over D.
- Let us define a function $SP : S \rightarrow \mathbb{N}$, which assigns each SI to a processor.
- Let $|S|_p$ be the number of statement instances assigned to processor p.
- Let us define a function $Sched : S \rightarrow |S|_{SP(s)}$, which assigns each $s_i \in S$ to a position t in the schedule of its currently assigned processor $SP(s_i)$.
- Let us define a function $SD : S \rightarrow D^*$, which assigns each $s_i \in S$ a sequence of DIs that are accessed by s_i. The order of the sequence is the access order.

- Let us define a function $DB : D \rightarrow B$, which assigns each DI to a block b in memory ($B = \{b_1, \ldots, b_{|B|}$).

- Let us define a *directed acyclic graph* dag with n nodes labelled $s_1, \ldots, s_n$. The dag represents the data dependencies in the following way: If there is a data dependency $s_i \delta s_j$, there has to be a path between s_i and s_j as well.

Now we can state our questions concerning the adaption of the local search algorithm more precisely. These are in particular:

What does one solution $f \in F$ look like?

The first question has been partially answered in section 2.3 already. The solutions we are searching for are nothing more than feasible program instances, consisting of $|S|$, $|S|_p$, $|D|$, SD, $Sched$ and a corresponding function SP. Using this function, the programmer should be able to notice regularities in the distribution of SIs across processors, making it possible to generalize the found solution to her problem. Examples of possible PIs are given in section 3.3.

What are the *problem specific constraints* that discriminate feasible solutions $f \in F$ from non–feasible ones?

There are two constraints to be obeyed. First, if two SIs s_i and s_k are scheduled on the same processor, and s_i is scheduled before s_k, there must not be a path from s_k to s_i in the dag. Or in other words: s_i can only be scheduled before s_k on the same processor, if s_i is not dependent on s_k (for a discussion of dependencies see section 2.2.6). One or more dependencies between s_i and s_k are allowed, if they are scheduled on different processors (e. g. $SP(s_i) = i, SP(s_k) = k, i \neq k$), since in this case the program execution on processor i can be halted, until s_k is finished on processor k. The objective function notices the dependency in this case, and adjusts its result accordingly (more on this in section 3.2.1).

There is an exception, however, and it constitutes the second constraint. Assume, s_1 is scheduled before s_2 on processor 1, and s_3 is scheduled before s_4 on processor 2. Now, if s_1 is dependent on s_4, and s_3 is dependent on s_2 (see figure 3.2), there is no way for the processors to finish their schedule, they are stuck in a dead lock. Solutions $f \in F$ that include some kind of cross dependencies are therefore not feasible.

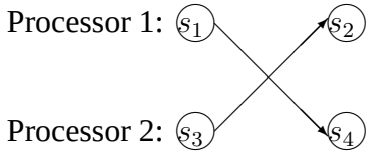


Figure 3.2: Representation of a cross dependency

How does one find an initial random solution $f \in F$?

To maximize internal reuse, the original random PI producing function of the `iblOpt` programming tool has been adapted for our needs. It produces a PI, which obeys our problem specific constraints, by repeatedly picking a random dag node $s_i \in S$ with indegree zero, and setting $Sched(s_i) = t$ and $SP(s_i) = 0$. t starts from zero and is incremented stepwise for each scheduled SI until it reaches $|S|$. Each s_i is removed from the dag after it has been scheduled, along with all its outgoing edges.

How is the neighborhood $NB(f)$ constructed?

The construction of a feasible neighbor $f_{new} \in F$ to a given $f \in F$ is accomplished in the following way:

1. On an arbitrary processor p, select a number k (with $0 < k \leq |S|_p$, where $|S|_p$ is the number of SIs currently scheduled on processor p) of successively scheduled SIs.

2. - Move the selected SIs to an earlier position in the schedule of the same processor p, where no dependencies for any SIs are violated, by adjusting $Sched$ accordingly and leaving SP as is for the selected entries.

   or

   - Move the selected SIs to an arbitrary position on a different processor q ($q \neq p$), where no dependencies for any SIs are violated, by adjusting $Sched$ accordingly and setting $SP(s_i) = q$ for all previously selected s_i.

   or

   - If possible and allowed by the dependencies, do the same steps as in one of the two previous points and additionally move k successively scheduled SIs, which are scheduled right before the insertion point, to the exact position, where the former ones used to be scheduled. This effectively makes the whole transaction an exchange operation (consisting of two moves).

The resulting f_{new} is a member of F, because the constraints of our problem domain are obeyed in every version of step 2.

What does the objective function look like?

The objective function and its subcomponents are described in section 3.2.

What are the inputs and the outputs of the final problem specific local search algorithm?

The algorithm takes the following inputs:

- The number of SIs $|S|$.
- The number of DIs $|D|$.
- A function $SD : S \rightarrow D^*$.
- A function $DB : D \rightarrow B$.
- A directed acyclic graph dag.
- Some basic memory hierarchy parameters including $C \in \mathbb{N}$ (*cache size*) and $L \in \mathbb{N}$ (*cache line size*).
- The number of processors n_p.
- The time it takes to access a value in the cache $t_{CHT} \in \mathbb{N}$ (*cache hit time*).
- The time it takes to access a value that is not in the cache and needs to be loaded from main memory $t_{MLT} \in \mathbb{N}$ (*memory load time*).
- The time it takes to access a value that is stored in the cache of a different processor, which has modified and not yet written the block back to main memory $t_{CST} \in \mathbb{N}$ (*cache sync time*).

The algorithm produces the following outputs:

- A function $Sched : S \rightarrow |S|_{SP(s)}$.
- A function $SP : S \rightarrow \mathbb{N}$.

What does the final problem specific local search algorithm look like?

The answers given for the questions above result in the pseudo–code algorithm presented in figure 3.3.

It is derived from the basic algorithm given in figure 3.1. In every iteration of the main loop, it tries to move a single group of successively scheduled SIs to a new position, possibly on a different processor. Afterwards, the objective function of the newly created PI is calculated. If the last move produced a PI with a better objective function value than the best one found before, the move and the objective function result are saved.

```
f := initial feasible solution (PI, SP);
repeat
   modified := false;
   for groupSize := 1 to max (|S|_p) do
      for sourceProc := 0 to n_p do
         for cutBefore := |S|_sourceProc to groupSize with cutBefore − − do
            for targetProc := 0 to n_p do
               for insertBefore := |S|_targetProc to groupSize
                     with insertBefore − − do
                  if last move not permitted by DAG then
                     /* exit the innermost loop */
                  endif;
                  /* move groupSize SIs from sourceProc to their new position
                     on processor targetProc */
                  /* calculate objective function using the current PI and SP
                     and save result in movePerf, if it is better than the best
                     one found before */
                  /* try an exchange, by moving the same number of SIs from right
                     before insertion point to the original location if allowed
                     by the DAG */
                  /* calculate objective function using the current PI and SP
                     and save result in exPerf, if it is better than the best
                     one found before */
                  /* undo the last move and exchange operation*/
               endfor;
            endfor;
         endfor;
   endfor;
   endfor;
   if better f was found by a simple move then
      /* redo the move leading to movePerf */
      modified := true;
   elif better f was found by an exchange then
      /* redo the exchange leading to exPerf */
      modified := true;
   endif;
until not(modified);
```

Figure 3.3: The local search algorithm used for automatic parallelization

In a second step, the algorithm moves an equal number of successively scheduled SIs that are located before the insertion point of the previous move, back to the original position of the SIs moved in the first place (if that is allowed by the dag). Effectively, this additional step completes the first move, making it an exchange operation. The objective function is calculated again, and if the exchange produced a PI with a better objective function value than the best one found before, it is saved for later reference (along with its objective function result for further comparisons).

The exchange operation has become necessary, since the local search algorithm often produced suboptimal solutions without it. It does not add new solutions to $NB^*(f)$ however, since it can be replaced by two simple moves. It allows significant performance improvements of the combined algorithm though, and therefore the exchange functionality is described here as well.

After the inner loops have been finished, the algorithm tests if a better solution f than the current one has been found. If this is the case, f is substituted by the newly found best neighbor f_{best}, and a new iteration cycle is started. If no better solution was found, the algorithm terminates and the $f \in F$, which was tested during the last turn is returned as a result.

The algorithm presented above is not yet optimized for performance. Significant performance improvements are possible, some of them are described in section 6.2.

3.2 The Objective Function

For our specific problem domain, the objective function $o(f)$ maps a PI f into the real numbers, thereby indicating the solution quality. We are interested in solutions that distribute all input SIs across the given number of processors well. But how is a good distribution distinguished from a bad one? As section 2.2.1 already stated, the most widely used measure for performance is *wall–clock time*. Above all, we are interested in solutions that perform as fast as possible on the given number of processors. Therefore, our objective function has a simulation component (described in more detail in 3.2.1) that calculates, among other things, an approximation of the time any given solution (PI) needs to access all required data through the memory subsystem (short: *memory–access time*). A more accurate measure for performance would be the wall–clock time of the program, but this would require the user to input an approximation of the execution time of every single SI of the input program, which would be inaccurate (since the user can only make an educated guess about these values) and would furthermore be quite time consuming, especially for large input PIs. Instead, we assume for simplicity that every SI has the same constant execution time. It remains a task for the programmer, to structure the input program accordingly. In our context, wall–clock time can be coarsely approximated by the following formulas:

$$t_{WallClock} = Max_{i=0}^{n_p} t_{WallClock_i} \tag{3.2}$$

$$t_{WallClock_i} = t_{SIExecution_i} + t_{MemoryAccess_i} \tag{3.3}$$

Above, the index i is used to indicate that the given times are calculated for each processor. The wall–clock time of the entire program is naturally equal to that of the slowest processor. Since the SI execution time remains a constant value, memory–access time is a sufficient approximation for wall–clock time. It is calculated in the following manner for our target SMP–architecture with one level of cache:

$$t_{MemoryAccess} = Max_{i=0}^{n_p} t_{MemoryAccess_i} \tag{3.4}$$

$$t_{MemoryAccess_i} = t_{CacheLoad_i} + t_{MainMemoryLoad_i} + t_{CacheSync_i} + t_{DependencyWait_i} \tag{3.5}$$

$t_{CacheLoad_i}$ stands for the time it takes to perform all cache accesses on processor i, while $t_{MainMemoryLoad_i}$ is the time required to load the remaining data items from main memory. $t_{CacheSync_i}$ is the time it takes for processor i to synchronize its cache with a different processor k. This becomes necessary, when write–back caches are employed, and a data item has been modified on processor k and not been written back to main memory. As soon as processor i wants to read the data item in question, it has to wait a certain time until the item has been delivered from processor k, the so called *cache sync time*. $t_{DependencyWait_i}$ marks the time, a processor has to wait for data dependencies of its instructions on other processors (see section 2.2.6 for details on dependencies). This time is usually spent on resynchronization constructs, like the ones described for OpenMP in section 2.2.5. For the reasons stated above, memory–access time is our first candidate for an objective function.

However, sometimes many solutions with an equal memory–access time have to be evaluated. Therefore, a secondary objective function was introduced. It is defined as follows:

$$t_{AccumulatedMemoryAccess} = \sum_{i=0}^{n_p} t_{MemoryAccess_i} \tag{3.6}$$

In other words, it is the sum of the memory–access times, across all processors. While the primary objective function is only able to detect improvements that directly affect the slowest unit, the secondary objective function notices improvements of the memory–access times across all processors. Since we want all units to perform as fast as possible (and the slowest processor must not remain that way after the next iteration), the *accumulated memory–access time* is also a well suited candidate for an objective function.

A tertiary objective function was added, which is defined in the following way:

$$variance = \frac{\sum_{i=0}^{n_p} (\frac{|S|}{n_p} - |S|_i)^2}{n_p} \tag{3.7}$$

This objective function turned out to be necessary during experiments, because sometimes the two functions described above were not sufficient to single out a best solution $f_{best} \in NB(f)$ for an iteration. Naturally, the SIs should be distributed across the processors as evenly as possible, with no single unit being significantly busier than another (especially, since we assume that all SIs take the same execution time to process). The well known mathematical reference number *variance* proved to be able to assure this.

All three objective functions are calculated and evaluated for every solution f. To allow two solutions f_1 and f_2 to be compared, a wrapper function around the objective functions was constructed. Its pseudo–code implementation is presented in figure 3.4.

if $t_{MemoryAccess}(f_1) < t_{MemoryAccess}(f_2)$ **then**
  return -1;
elif $t_{MemoryAccess}(f_1) > t_{MemoryAccess}(f_2)$ **then**
  return 1;
elif $t_{AccumulatedMemoryAccess}(f_1) < t_{AccumulatedMemoryAccess}(f_2)$ **then**
  return -1;
elif $t_{AccumulatedMemoryAccess}(f_1) > t_{AccumulatedMemoryAccess}(f_2)$ **then**
  return 1;
elif $variance(f_1) < variance(f_2)$ **then**
  return -1;
elif $variance(f_1) > variance(f_2)$ **then**
  return 1;
else return 0;

Figure 3.4: The wrapper objective function

The function returns -1, if f_1 is the better solution and 1 if f_2 is to be preferred. If the wrapper function cannot determine a clear winner, it returns 0. Memory–access time is the primary criterion for evaluation, followed by accumulated memory–access time and variance. Using these three components, the wrapper objective function is usually able to narrow down the neighborhood $NB(f)$ to a few equally good solutions for each iteration. It is not perfect yet, though, and an idea for improvement is sketched in section 6.2.

3.2.1 The Runtime Simulator

The most substantial part of the objective function is the simulation of program execution and in particular memory–accesses for a candidate solution f. At present, the simulation refers to an SMP architecture with the following cache characteristics (for descriptions of these see sections 2.1.1 and 2.2.3):

- fully associative
- least recently used
- write back
- write allocate
- write invalidate
- cache coherent

It should be relatively straightforward to generalize the implemented function to different cache architectures when needed (we think, however that minor architectural changes would not have a big impact on the solution f, since the simulation is just a coarse one anyways).

The actual simulation is accomplished by stepwise and parallel simulation of the memory–accesses needed for each SI on every processor, while at the same time keeping track of the current contents of the different caches. When dependencies are encountered, program execution on a processor can be halted, until the dependency has finished executing on a different processor (dependencies, which wait for instructions on the same processor are not allowed, see section 3.1 for details).

For a good approximation, the following parameters (introduced in section 3.1 already) are needed: C, L, t_{CHT}, t_{MLT} and t_{CST}. These must be guessed by the programmer; the exact values of the timings are not important, only the ratio t_{CHT}:t_{MLT}:t_{CST} matters, which is used in the function to calculate an overall memory–access time. We think that the ones used throughout this thesis and presented in section 5 should be sufficient to recognize the parallelism inherent in many programs. Furthermore, no exact numbers are needed, approximations will be good enough most of the time. It is advisable, to round t_{MLT} and t_{CST} to multiples of t_{CHT}, to cut down the time it takes to process the simulation function.

A pseudo–code implementation of the simulation function is given in figure 3.5.

3.3 Suitable Examples

Three examples have been chosen for this thesis to present the capabilities of the modified `iblOpt` programming tool. All of them are well known in the literature, and have been thoroughly examined by compiler writers, compilers and parallelization experts for potential of parallelization. Therefore, we can compare the parallel versions of these programs derived from our tool to various different results presented in the literature.

In this section, each example program is explained shortly, along with the reasons for choosing it. A sequential implementation of every program (witten in `ANSI C`) completes the picture and is presented in appendix A.

```
currentTime := 0;
for proc := 0 to n_p do
    currentSI[proc] := first instruction in schedule for proc;
    nextFreeTime[proc] := 0;
endfor;
repeat
    done := true;
    for proc := 0 to n_p do
        if currentSI[proc] = 0 then /* continue with next processor */
        if (currentTime < nextFreeTime[proc]) or
            (an instruction, on which this SI depends, is not yet finished) then
            done := false;
            /* continue with next processor */
        endif;
        foreach block := DB(SD(currentSI[proc])) do
            if block is found in cache of processor proc then
                /* mark block as being least recently used */
                nextFreeTime[proc] := nextFreeTime[proc] + t_CHT;
            elif block is found in cache of processor different from proc then
                if block in foreign cache is not synchronized to main memory then
                    nextFreeTime[proc] := nextFreeTime[proc] + t_CST;
                else nextFreeTime[proc] := nextFreeTime[proc] + t_MLT;
                if write access to block should be performed then
                    /* declare block in remote cache invalid (write invalidate) */
                endif;
                /* copy block into local cache, while dropping least recently used entry */
            elif block is found in main memory only then
                /* copy block into local cache, while dropping least recently used entry */
                nextFreeTime[proc] := nextFreeTime[proc] + t_MLT;
            endif;
        endforeach;
        if next(currentSI[proc]) <> 0 then done := false;
        currentSI[proc] := next(currentSI[proc]);
    endfor;
    currentTime := next(currentTime);
until done = true;
```

Figure 3.5: The runtime simulation function

Matrix Transposition

Matrix transposition is one of the standard operations of linear algebra, and described in many books about mathematics, e. g. [Zei96]. It takes a matrix $A \in Mat(m, n)$ as input, and the resulting output matrix A^T is one of type (n, m). A^T is derived from A by exchanging the rows and columns of the input matrix. A sequential version of the transposition algorithm is given in appendix A.1.

The program does not have any dependencies in its code, so it is a relatively easy target for parallelization. It was chosen because it is well explained in the literature, and many techniques for exploiting parallelism have been successfully used on it.

Matrix Multiplication

Matrix multiplication is another frequently used operation in linear algebra and described in many books, e. g. ([Zei96]). It takes a matrix $A \in Mat(m, n)$ and a matrix $B \in Mat(n, p)$ as input, and results in a matrix $C \in Mat(m, p)$, whose elements are calculated like this:

$$c_{jk} = \sum_{s=1}^{n} a_{js} * b_{sk} \tag{3.8}$$

For the sake of simplicity (and because it introduces no fundamental changes to the algorithm), only matrices of the form $A, B \in Mat(n, n)$ are accepted as input, and therefore the output of our example program is a matrix of the form $C \in Mat(n, n)$ also. A sequential version of the multiplication algorithm is given in appendix A.2.

Like matrix transposition, the matrix multiplication program does not have dependencies in its code, thus the same reasons for choosing it apply. What makes it special and worth describing in detail is its greater complexity, there are three matrices involved instead of two, and a lot more calculations are taking place.

The Laplace Equation

The laplace equation is a partial differential equation, and also described in many publications (e. g. [Lap03]). The equation is important in many scientific fields, among them astronomy and electromagnetism. The equation is written as follows:

$$\bigtriangledown^2 \phi = 0 \tag{3.9}$$

There is a relatively simple way of solving the laplace equation by using a method commonly known as *Jacobi Iteration*. Iterative methods always improve an existing approximation, and this one is no exception. The algorithm alters every element of a matrix, by calculating the average value of its four neighbors for every iteration. The updates happen at once, so the old

values of the neighbors are always used. Algorithms with these characteristics are also called *stencil codes*. A sequential version is given in appendix A.3.

As one can clearly see in the source code, this example includes internal dependencies. The statements in the `for`–loop starting on line 54 must be processed after the ones in the previous `for`–loop. Dependencies must also be obeyed between iterations of the main loop (on line 46), which makes this example more difficult to optimize.

Chapter 4

Implementation

This chapter gives a short overview of the implementation details of our work on the `iblOpt` programming tool, down to an analysis of the files that were added or heavily modified by us. Since the tool is implemented using the `C` programming language, this part of our work can be divided into a section about the header files, where all of our data structures and function declarations reside (4.1), and a section about the actual implementation of the functionality (4.2).

4.1 Header Files

This section documents the header files modified during the course of this thesis in alphabetical order, with a special emphasis on the modifications necessary for parallelization.

`DebugHelper.h`

Holds declarations for the debugging functions contained in `DebugHelper.c`.

`Grundtypen.h`

Contains the most important data structures related to SI-, DI- and PI–Management, as well as various constant definitions. It is possibly the most important header of the entire project, and a good starting point for getting familiar with the source code. Our main contribution to this file is the `paraSchedDataStruct` data structure, and the `ParaSchedDataTyp` data type associated with it. Therein, parallelization parameters for every statement instance (e. g. the processor on which the SI is to be scheduled) are contained.

LoadTime.h

Contains the `CacheEntryTyp` data type that holds all required data to represent a block in our simulated cache. It is heavily used by the runtime simulator function described in 3.2.1. Moreover, it contains function declarations for `LoadTime.c`.

Parallel.h

Is home of the `piPerfTyp` data type. The performance parameters of every evaluated program instance are stored in a data structure of this type, which makes it indespensable for the calculation of the objective function (described in detail in section 3.2). Furthermore, it holds declarations for the functions contained in `Parallel.c`.

SIMove.h

Could be named `SIGroupMove.h`, since it contains the necessary data types to accomplish the move and replace operations of connected groups of SIs, extensively described in section 3.1. It also has declarations for the functions contained in `SIMove.c`.

4.2 Program Files

This section documents the `C`–source files modified during the course of this thesis in alphabetical order, with a special emphasis on the modifications necessary for parallelization. Minor changes on some files were omitted, if they contributed nothing to the general understanding of the implementation.

DebugHelper.c

Contains functions used for debugging purposes during program development. Among them are `printSchedules` (prints out the current schedules for all processors), `printMove` (outputs the characteristic parameters of a move of a group of SIs), `printResults` (writes PI performance data to the screen) and `printDAG` (prints out the *directed acyclic graph* dag that holds the dependency relations between statement instances).

LoadTime.c

Includes only two functions, one of which is `calcLoadTime`. It implements the runtime simulator function, whose ideas and structure are sketched in 3.2.1. The second function is called `wrapCalcLoadTime` and it implements an additional layer around the former, which

allows all optimization functions contained in `iblOpt` to utilize it, even the sequential ones. This feature is useful for performance comparisons.

`Optimierung_Para1.c`

Holds the implementation of the `Optimiere_Para1` function. It is possibly the most important function of the whole program, as all the ideas presented in section 3.1 concerning the optimization algorithm are realized there.

`Parallel.c`

Contains some basic helper functions for the parallelizing part of the `iblOpt` program, including e. g. memory allocations and deallocations in initialization and cleanup functions. Furthermore, the objective function described in 3.2 is implemented here (called `paraTargetFunc`), along with an extended dependency checking function called `isPredecessor`.

`SIMove.c`

Includes the function necessary to move groups of SIs from one position in the schedule to a different one, possibly even onto a new processor (called `doMove`). The functionality to move SIs back to their original position is also contained here, located in the function `undoMove`. Both involve many list operations, and since these are implemented using pointers in `C`, the functions are small, but error prone.

Chapter 5

Results

In this chapter we present some results obtained by using the semi–automatic method, including the modified `iblOpt` programming tool, for parallelization. This is accomplished by walking through the steps involved in parallelizing the example programs presented in section 3.3. For the matrix transposition example of section 5.1, everything is described in great detail, while for the rest of the programs only important differences are noted.

The steps involved in parallelization do not differ too much from the original semi–automatic method, which consists of three main phases (*instantiation phase*, *optimization phase* and *generalization phase*), as described in more detail in section 2.3 and [Leo01c].

5.1 Matrix Transposition

The sequential version of the matrix transposition program is given in appendix A.1. In the instantiation phase it needs to be converted to a program instance (PI). Since the `iblOpt` programming tool is not able to read plain `C` source code, the program must be slightly altered. Furthermore, in the instantiation phase, parameters for input, memory hierarchy and parallelization need to be specified. Usually, it is a good idea to keep the input size as small as possible, so we are going to perform our experiments with an input size of $N = 4$ (additional measurements for an input size of $N = 6$ are also included).

With an input size that small, we need to restrict the used *cache size* C and *cache line size* L, to keep it proportional to a real world program. After a little experimenting, we found that C = 4 and L = 2 were sufficient and achieved good results for this example.

For parallelization parameters, we are concentrating on the case that the example program is distributed across two processors, additional results for one and four processors are included. The one processor case is particularly useful to compare our modified algorithm to the original locality optimization algorithm (used as a reference for performance later).

The memory timing parameters t_{CHT}, t_{MLT} and t_{CST} have been fixed at a ratio of 100 :

$1000 : 2000$. Remember, as explained in section 3.2.1 that these are not really measured in units of time like ms or ns, but are merely indications of e. g. how long it takes to access one level of memory compared to the next slower level (e. g. a cache compared to main memory), therefore a ratio of $1 : 10 : 20$ would work just the same. These values perform fine for our simulated memory architecture (also explained in detail in 3.2.1).

Based on these input parameters, an initial PI and a dag are automatically constructed. Since the program does not have internal dependencies, the dag is empty and omitted here. Therefore, the PI consists of a random variation of the SIs involved, which are all scheduled on one processor (see section 3.1).

In the next phase, the optimization algorithm is applied (optimization phase). It results in an output PI, along with some performance measures for the given solution. These include an approximate overall memory–access time t_{OVL} (which, again, is only useful for comparisons and no indication of runtimes on existing architectures) and the number of *cache misses*, *cache synchronizations* and *memory loads*.

For our example, the performance measures are given in table 5.1. The table also includes performance measures from our reference called *Memory Loads Reference* and t_{OVL} *Reference* (obtained by using the original `iblOpt` algorithm, which of course only works on a single processor). The speedups are calculated using the formula introduced in section 2.2.1.

Input size	4	4	6	6	6
Processors	1	2	1	2	4
C	4	4	9	9	9
L	2	2	3	3	3
Cache Hits	12	12	32	32	32
Cache Syncs	0	0	0	0	0
Memory Loads	20	20	40	40	40
t_{OVL}	21200	10600	43200	21600	10800
Memory Loads Reference	20	20	40	40	40
t_{OVL} Reference	21200	21200	43200	43200	43200
Speedup	1.00	2.00	1.00	2.00	4.00

Table 5.1: `iblOpt` results for matrix transposition

As one can see in the table, a significant potential for parallelization seems to be inherent in the matrix transposition example. The algorithm manages to cut down the simulated overall memory–access time proportional to the number of processors used. No cache synchronizations need to take place (since there are no dependencies). Also note that our parallel algorithm, applied to one processor, performs equal to the reference implementation (for both performance measures t_{OVL} and number of memory loads), which indicates that the used runtime simulation function is doing a good job optimizing for locality as well.

Now let us take a look at an output PI of the algorithm (for the case of N = 4, distribution

onto two processors), which is presented in figure 5.1.

SI	Proc.	EndTime	SI	Proc.	EndTime
A[1,2]=B[2,1]	0	2000	A[3,0]=B[0,3]	1	2000
A[0,2]=B[2,0]	0	3100	A[3,1]=B[1,3]	1	3100
A[0,3]=B[3,0]	0	4200	A[2,1]=B[1,2]	1	4200
A[1,3]=B[3,1]	0	5300	A[2,0]=B[0,2]	1	5300
A[1,0]=B[0,1]	0	7300	A[3,3]=B[3,3]	1	7300
A[1,1]=B[1,1]	0	8400	A[3,2]=B[2,3]	1	8400
A[0,1]=B[1,0]	0	9500	A[2,2]=B[2,2]	1	9500
A[0,0]=B[0,0]	0	10600	A[2,3]=B[3,2]	1	10600

Figure 5.1: An optimized PI for matrix transposition with N = 4 on two processors

In this figure, the first column has the scheduled SIs, the second shows on which processor the SI is scheduled (and is therefore equal to the result of the function SP, introduced in section 3.1) and the third one indicates the simulated time, when the SI has finished all memory–accesses. This value is especially important, when there are dependencies to be obeyed (which is not the case for this example), because it shows where synchronization barriers need to be placed by the programmer.

The most difficult part of the semi–automatic method remains: the generalization phase. Now the programmer has to notice patterns in the output PIs, which enable parallelization. They are usually not obvious from looking at one output PI, sometimes one has to consider many of them to recognize the patterns. In our example, there is one striking fact to notice: On processor number zero, all memory–accesses of array A have the form $A[0, x]$ or $A[1, x]$, where $0 <= x <= 3$. Similarly, on processor number one, all memory–accesses of A look like this: $A[2, x]$ or $A[3, x]$, where $0 <= x <= 3$.

This is not a coincidence, but a strong indication that a parallelization technique called *loop tiling* is able to help in distributing the program. Wolfe defines *loop tiling* like this:

> *Loop tiling: A compiler transformation that organizes a set of nested loops to work on small parts of the iteration space, to take advantage of locality.*
>
> ([Wol96, p. 538])

By dividing the input matrix A into two submatrices and processing them on different processors, the observed speedups should be possible to reach in a real world program, too. At the same time, we are able to use the principle of locality to further speed up the program, because only accesses to half of the array A are necessary on each processor.

These suggestions are put into practice in the `trapopara.c` program, which is given in appendix B.1. It is also written in `ANSI C` and uses `OpenMP` directives to achieve parallel execution. It was compiled and run on the `Hessischer Hochleistungsrechner --`

HHLR cluster, which consists of several computers of the IBM eServer pSeries class, with a total of 104 processors using the AIX operating system (see [Sch03] for details). As one can see, a single OpenMP directive is sufficient to achieve the intended effect.

Note that for locality optimization, it would be better to not only split the matrix vertically (like it is done in our example, this special case of loop tiling is called *strip mining*), but also horizontally. The iblOpt tool clearly shows this, when optimizing for e. g. four processors. To implement it in OpenMP however, nested directives or explicit work distribution would need to be utilized, which would increase control overhead or synchronization costs. Our experiments showed that the simple version used in B.1 performed better, and therefore we chose it for our comparisons.

Finally, let us see if we were right, and the program modifications led to a decrease in program execution time. The timing results of the experiments with the trapopara.c program are presented in table 5.2.

Input size	1000	2000	4000	1000	2000	4000	1000	2000	4000
Processors	2	2	2	4	4	4	8	8	8
W.–clock t.	8.45	65.70	327.94	4.14	33.03	166.69	2.18	22.14	239.15
Ref. w.–clock t.	16.79	131.25	653.74	16.79	131.25	653.74	16.79	131.25	653.74
Speedup	1.99	2.00	1.99	4.06	3.97	3.92	7.70	5.92	2.73

Table 5.2: Timing results for the parallel matrix transposition program

The sequential version traposeq.c was used as a reference implementation here. The abbreviation *W.–clock t.* stands for wall–clock time and *Ref. w.–clock t.* serves as a placeholder for reference wall–clock time. All times are in seconds and were measured from within the program.

Note that the numbers can be only approximations, since there were other programs running on the machine during the time measurements as well, which might have spoiled the results. Especially the measured value for $N = 4000$ on eight processors seems a little high, as it is even slower than the four processor case.

As one can still see from these numbers, the speedups are about proportional to the number of processors used, which leads to an efficiency value (see section 2.2.1) around 1.00 for this algorithm and implementation (even a superlinear speedup was achieved for one example, for an explanation see 2.2.1). Therefore, obviously the parallelization for this program works as expected.

5.2 Matrix Multiplication

The sequential version of the matrix multiplication program is given in appendix A.2. The same thoughts about choosing the required parameters as in the previous section 5.1 apply here, an

input size of $N = 4$ is used, along with the very same memory timing parameters t_{CHT}, t_{MLT} and t_{CST} in a ratio of $100 : 1000 : 2000$. The only value that was slightly increased to eight was the cache size C, because of the additional matrix that needs to be processed simultaneously (the cache line size L remained at a value of two). Since the matrix multiplication algorithm has no dependencies as well (and therefore an empty dag), the starting PI consists of a random variation of the SIs involved, all scheduled on the first processor. The optimization results presented in table 5.3 were obtained for the described setup.

Input size	4	4	4
Processors	1	2	4
C	8	8	8
L	2	2	2
Cache Hits	143	142	140
Cache Syncs	0	0	0
Memory Loads	49	50	52
t_{OVL}	63300	32100	16500
Memory Loads Reference	49	49	49
t_{OVL} Reference	63300	63300	63300
Speedup	1.00	1.97	3.83

Table 5.3: `iblOpt` results for matrix multiplication

Similar to matrix transposition, a significant potential for parallelization seems to exist here as well (which is not surprising, since there are no dependencies to be obeyed). The parallel algorithm applied to the one processor case again manages to achieve results equal to our reference. However, the speedups obtained for the two and four processor case are slightly below the optimum. This is to be expected, as there are more compulsory misses happening here (see section 2.1.1), which slightly spoil the result. Since the performance measures are just indications at this point anyways, let us move on to take a look at the output PI, presented in figure 5.2.

As in the previous example, a pattern is clearly recognizable: On processor number zero, all accesses to matrix C have the form: $C[0, X]$ or $C[2, X]$, where $0 <= X <= 3$. Similar access patterns can be observed on processor number one, which is a strong indication that *strip mining* will be able to help us in parallelizing the problem. With these thoughts in mind, the `multpara.c` program has been written, its source code is presented in appendix B.2. Note that there were output PIs having $C[0, X]$ and $C[1, X]$ scheduled on the same processor as well, so in our optimized example a blocking approach was used for simplicity. The results achieved on the `HHLR` can be observed in table 5.4.

Note that we were not able to obtain any results for an input size of $N = 4000$, since three arrays of that size would not fit into the available memory of the test–machine.

As one can see from the timing values given, a speedup proportional to the number of used

SI	Proc.	EndTime	SI	Proc.	EndTime
C[2,0]+=A[2,2]*B[2,0]	0	3000	C[1,3]+=A[1,2]*B[2,3]	1	3000
C[2,1]+=A[2,3]*B[3,1]	0	4200	C[1,3]+=A[1,3]*B[3,3]	1	4200
C[2,0]+=A[2,3]*B[3,0]	0	4500	C[1,2]+=A[1,2]*B[2,2]	1	4500
C[2,1]+=A[2,2]*B[2,1]	0	4800	C[1,2]+=A[1,3]*B[3,2]	1	4800
C[0,1]+=A[0,2]*B[2,1]	0	6900	C[1,2]+=A[1,1]*B[1,2]	1	6900
C[0,0]+=A[0,2]*B[2,0]	0	7200	C[1,3]+=A[1,0]*B[0,3]	1	8100
C[0,1]+=A[0,3]*B[3,1]	0	8400	C[1,2]+=A[1,0]*B[0,2]	1	8400
C[0,0]+=A[0,3]*B[3,0]	0	8700	C[1,3]+=A[1,1]*B[1,3]	1	8700
C[0,3]+=A[0,3]*B[3,3]	0	10800	C[3,3]+=A[3,1]*B[1,3]	1	10800
C[0,2]+=A[0,3]*B[3,2]	0	11100	C[3,2]+=A[3,1]*B[1,2]	1	11100
C[0,3]+=A[0,2]*B[2,3]	0	12300	C[3,2]+=A[3,0]*B[0,2]	1	12300
C[0,2]+=A[0,2]*B[2,2]	0	12600	C[3,3]+=A[3,0]*B[0,3]	1	12600
C[2,3]+=A[2,2]*B[2,3]	0	14700	C[3,0]+=A[3,0]*B[0,0]	1	14700
C[2,3]+=A[2,3]*B[3,3]	0	15900	C[3,1]+=A[3,1]*B[1,1]	1	15900
C[2,2]+=A[2,3]*B[3,2]	0	16200	C[3,0]+=A[3,1]*B[1,0]	1	16200
C[2,2]+=A[2,2]*B[2,2]	0	16500	C[3,1]+=A[3,0]*B[0,1]	1	16500
C[2,3]+=A[2,0]*B[0,3]	0	18600	C[1,1]+=A[1,0]*B[0,1]	1	18600
C[2,2]+=A[2,0]*B[0,2]	0	18900	C[1,1]+=A[1,1]*B[1,1]	1	19800
C[2,2]+=A[2,1]*B[1,2]	0	20100	C[1,0]+=A[1,0]*B[0,0]	1	20100
C[2,3]+=A[2,1]*B[1,3]	0	20400	C[1,0]+=A[1,1]*B[1,0]	1	20400
C[2,1]+=A[2,0]*B[0,1]	0	22500	C[1,0]+=A[1,2]*B[2,0]	1	22500
C[2,0]+=A[2,0]*B[0,0]	0	22800	C[1,1]+=A[1,2]*B[2,1]	1	22800
C[2,0]+=A[2,1]*B[1,0]	0	24000	C[1,1]+=A[1,3]*B[3,1]	1	24000
C[2,1]+=A[2,1]*B[1,1]	0	24300	C[1,0]+=A[1,3]*B[3,0]	1	24300
C[0,1]+=A[0,1]*B[1,1]	0	26400	C[3,1]+=A[3,3]*B[3,1]	1	26400
C[0,1]+=A[0,0]*B[0,1]	0	27600	C[3,0]+=A[3,3]*B[3,0]	1	26700
C[0,0]+=A[0,1]*B[1,0]	0	27900	C[3,1]+=A[3,2]*B[2,1]	1	27900
C[0,0]+=A[0,0]*B[0,0]	0	28200	C[3,0]+=A[3,2]*B[2,0]	1	28200
C[0,2]+=A[0,1]*B[1,2]	0	30300	C[3,3]+=A[3,3]*B[3,3]	1	30300
C[0,3]+=A[0,0]*B[0,3]	0	31500	C[3,2]+=A[3,2]*B[2,2]	1	31500
C[0,2]+=A[0,0]*B[0,2]	0	31800	C[3,2]+=A[3,3]*B[3,2]	1	31800
C[0,3]+=A[0,1]*B[1,3]	0	32100	C[3,3]+=A[3,2]*B[2,3]	1	32100

Figure 5.2: An optimized PI for matrix multiplication with $N = 4$ on two processors

Input size	1000	2000	1000	2000	1000	2000
Processors	2	2	4	4	8	8
W.–clock t.	9.89	178.36	5.03	78.90	2.52	53.35
Ref. w.–clock t.	20.38	313.89	20.38	313.89	20.38	313.89
Speedup	2.06	1.76	4.05	3.98	8.09	5.88

Table 5.4: Timing results for the parallel matrix multiplication program

processors is achieved, thereby allowing efficiency results of about 1.00, an indication that the technique used for parallelization is worth applying.

5.3 The Laplace Equation

The sequential version of the laplace equation solver program is given in appendix A.3. The input size for our example is $N = 4$, the memory timing parameter ratio of $t_{CHT} : t_{MLT} : t_{CST} = 100 : 1000 : 2000$ stays untouched as well. Cache size C for our example is 8, and cache line size is 2.

The laplace equation solver program does have dependencies, as outlined in section 3.3. Therefore, the starting PI is not a random variation of SIs, but rather one that obeys the dependency rules outlined in section 3.1. The optimization phase of this example leads to the results presented in table 5.5.

Input size	4	4	4
Processors	1	2	4
C	8	8	8
L	2	2	2
Cache Hits	26	30	29
Cache Syncs	0	2	4
Memory Loads	58	52	51
t_{OVL}	60600	29500	16300
Memory Loads Reference	58	58	58
t_{OVL} Reference	60600	60600	60600
Speedup	1.00	2.05	3.72

Table 5.5: `iblOpt` results for the laplace equation

In spite of the dependencies, high speedups (even superlinear for the two processor case) are achieved by `iblOpt`. An output PI is presented in figure 5.3.

Note that the PI shown here is not the one with the lowest calculated overall memory–access time (it has 16700, the best one found by `iblOpt` has 16300), but one where the parallelizing potential shows clearest. Most of the time, it is not sufficient to look at the fastest solution to

SI	Proc.	EndTime
B[2,1]:=A[2,1]	0	2000
A[2,1]:=0.25*(B[3,1]+B[1,1]+B[2,2]+B[2,0])	0	8100
B[2,1]:=A[2,1]	0	8300
A[2,1]:=0.25*(B[3,1]+B[1,1]+B[2,2]+B[2,0])	0	12400
B[2,1]:=A[2,1]	0	12600
A[2,1]:=0.25*(B[3,1]+B[1,1]+B[2,2]+B[2,0])	0	16700
B[1,2]:=A[1,2]	1	2000
A[1,2]:=0.25*(B[2,2]+B[0,2]+B[1,3]+B[1,1])	1	6100
B[1,1]:=A[1,1]	1	8200
A[1,2]:=0.25*(B[2,2]+B[0,2]+B[1,3]+B[1,1])	1	12300
B[1,2]:=A[1,2]	1	12500
A[1,2]:=0.25*(B[2,2]+B[0,2]+B[1,3]+B[1,1])	1	15700
B[2,2]:=A[2,2]	2	2000
A[2,2]:=0.25*(B[3,2]+B[1,2]+B[2,3]+B[2,1])	2	8100
B[2,2]:=A[2,2]	2	8300
A[2,2]:=0.25*(B[3,2]+B[1,2]+B[2,3]+B[2,1])	2	12400
B[2,2]:=A[2,2]	2	12600
A[2,2]:=0.25*(B[3,2]+B[1,2]+B[2,3]+B[2,1])	2	16700
B[1,1]:=A[1,1]	3	2000
A[1,1]:=0.25*(B[2,1]+B[0,1]+B[1,2]+B[1,0])	3	6100
B[1,2]:=A[1,2]	3	8200
A[1,1]:=0.25*(B[2,1]+B[0,1]+B[1,2]+B[1,0])	3	11400
B[1,1]:=A[1,1]	3	11600
A[1,1]:=0.25*(B[2,1]+B[0,1]+B[1,2]+B[1,0])	3	15900

Figure 5.3: An optimized PI for the laplace equation with $N = 4$ on four processors

recognize patterns. The programmer should rather look at the best set of solutions for optimal results.

It is obvious from looking at this example PI that loop tiling is able to help parallelizing this example program, despite of the dependencies. We chose to implement the special case strip mining once again, the same reasons for doing so as in the matrix transpostion program apply. A parallel version of the laplace equation solver problem was implemented, and can be found in appendix B.3. But are we able to achieve satisfying speedups in a real world program using this technique? The answer is given in table 5.6, where our results on the `Hessischer Hochleistungsrechner` are shown.

Input size	1000	2000	4000	1000	2000	4000	1000	2000	4000
Processors	2	2	2	4	4	4	8	8	8
W.–clock t.	4.85	19.35	79.91	2.44	10.27	42.28	1.32	5.84	23.68
Ref. w.–clock t.	9.88	38.24	178.38	9.88	38.24	178.38	9.88	38.24	178.38
Speedup	2.04	1.98	2.23	4.05	3.72	4.22	7.48	6.55	7.53

Table 5.6: Timing results for the parallel laplace equation program

Despite the synchronization costs, the speedups here were as high as in the programs without dependencies (even some relatively high superlinear ones were measured). Note that these are only approximations though, as the machine was not fully unloaded during our tests. The parallelization techniques seem to pay off, as efficiency values of more than 1.00 are guaranteed to make parallelization worthwile.

Chapter 6

Conclusions

6.1 Summary

The main accomplishment of this thesis is the enhancement of the semi–automatic method and its main software component, the `iblOpt` programming tool, to include automatic parallelization. Since `iblOpt` relied heavily on a locality optimizing algorithm based on a local search approach, the automatically parallelizing function was to use some form of local search as well.

To accomplish this task, an objective function based on three components (memory–access time, accumulated memory–access time, variance of the distribution of statements across processors) was defined, and in a later step implemented and tested in `iblOpt`.

For this approach to work reliably, it became necessary to design and implement a *runtime simulator* component, which was able to accurately simulate the memory–accesses performed by each statement on every processor, while at the same time keeping track of corresponding cache–contents. Using this functionality, it was possible to reliably calculate all necessary components of the objective function.

The second contribution of this thesis is the actual algorithm used for automatic parallelization in `iblOpt`, which was adapted for our task from the original locality optimizing solution. It contains the search for a specially constructed neighborhood, and is executed repeatedly for better results. It was shown that this algorithm was able to perform automatic parallelization on a number of sequential programs.

6.2 Outlook

Improvements and further work could be done on several possible subsystems of the `iblOpt` programming tool. Performance improvements of the search time for solutions would be the first sector that came to our mind. Presently, the local search algorithm evaluates far too many possible neighbors, before reaching a conclusion about what move or exchange operation to

perform next. In our example sessions for the matrix multiplication example, more than one million neighbors were evaluated, which led to running times of several minutes on an AMD Athlon XP 1700+ computer. For more complex examples, this could prove to be a problem. Whether or not it is really necessary to check that many neighbors, or if maybe a subset of the current neighborhood would be sufficient for equal results, remains subject to further research.

Performance also suffers from the runtime simulator function, which presently needs to be recalculated every time a neighbor is evaluated. An incremental solution that builds upon the performance measures obtained for a $f \in F$ to calculate these values for $f_k \in NB(f)$ would be much preferred.

Besides the issues mentioned with the runtime simulator function, the objective functions have potential for further tuning as well. Presently, cases remain, in which two possible neighbors $f_1, f_2 \in NB(f)$ are compared and no best solution can be established. Adding more objective functions could change this situation and possibly lead to better results. An obvious additional candidate for inclusion here is the OFL_{loc} function presented in [Leo98] which, even though it is only useful for locality optimization, is probably a good choice for an additional objective function.

Moving away from performance issues, there are a couple of areas where further work could start as well. Regularity optimization (explained in detail in [Leo01b]) is still not implemented for our parallelizing part of `iblOpt`, although this could probably be changed relatively fast by adapting the functionality of the single processor case. This change has the potential to greatly help the programmer during the generalization phase, where it is often quite difficult to recognize patterns from the output PIs.

The biggest improvements of the results could probably be achieved by not only optimizing statements for locality, but also data. This is implemented in the original `iblOpt` programming tool and documented e. g. in [Leo98], but has not been ported to our new parallelizing component yet. A whole new class of solutions could possibly be reached then.

Further research could also be done while changing fundamental assumptions. A relatively easy task would be adjusting the runtime simulator function to simulate different cache characteristics, e. g. *write–update* or *write–through*. Even though we do not expect great improvements of the results from changesets like this, they might be in order when an algorithm is supposed to be optimized for a special parallel architecture. In this case, tuning the memory–access timing parameters t_{CHT}, t_{MLT} and t_{CST} could also prove to be beneficial.

Going one step further, the concept of automatic parallelization using the semi–automatic method is not limited to the parallel architecture it is optimized for now, which is SMP. The next logical step in this direction would include extending the program to support NUMA–architectures. This task would involve at least introducing a third memory level, since besides caches and local main memory, one would also need to simulate the remote main memory (for details about NUMA–architectures see section 2.2.3). Of course, the memory–access timing parameters would have to be adjusted as well, along with the runtime simulator function, since

local memory does not behave like a cache at all.

Another possible architectural change worth pursuing could be `SIMD` architectures. The runtime simulator function would need to be rewritten of course, but on principle the general approach of the semi–automatic method should be able to cope with such a radically different architecture as well.

Besides the changes mentioned above, many more could be thought of and implemented, thus there are more than enough fields for further research.

Appendix A

Sequential Program Sources

A.1 traposeq.c

```
001 /* This is the sequential version of the Trapo program, used in
002  * iblOpt for demonstration purposes. It basically transforms the
003  * Matrix B and saves the result into Matrix A.
004
005  * Inputs (through defines in the Makefile):
006  *   N - matrix size
007
008  * Outputs:
009  *   the sum of all array elements
010  *   wall-clock time of the calculation phase
011
012  * Author: Michael Süß
013  */
014
015 /* needed for calculating seconds from nanoseconds */
016 #define BILLION  1000000000L;
017
018 #include <stdio.h>
019 #include <stdlib.h>
020 #include <time.h>
021
022 int A[N][N];
023 int B[N][N];
024
025 int main()
026 {
027   int i, j, k;
028   long sum = 0;
```

```
029   struct timespec start, stop;
030   double accTime;
031
032   /* Array initialization phase */
033   for (i = 0; i < N; i++)
034     for (j = 0; j < N; j++)
035       B[i][j] = i;
036
037   /* Start measuring the time */
038   clock_gettime(CLOCK_REALTIME, &start);
039
040   /* Calculation phase */
041   /* Timing loop, to make wall-clock-time more measurable */
042   for (k = 0; k < 1000; k++)
043     for (i = 0; i < N; i++) {
044       for (j = 0; j < N; j++)
045         A[i][j] = B[j][i];
046     }
047
048   /* Finish time measurement */
049   clock_gettime(CLOCK_REALTIME, &stop);
050
051   accTime =
052     (stop.tv_sec - start.tv_sec) + (double) (stop.tv_nsec -
053                           start.tv_nsec)
054     / (double) BILLION;
055
056   /* Verification phase */
057   for (i = 0; i < N; i++) {
058     for (j = 0; j < N; j++)
059       sum += A[i][j];
060   }
061
062   /* print out the sum of all array elements */
063   printf("sum: %ld\n", sum);
064
065   /* print out the wall-clock time of the calculation phase */
066   printf("wall-clock time = %.2lf (N=%d, P=1)\n", accTime, N);
067
068   return EXIT_SUCCESS;
069 }
070
071
```

A.2 multseq.c

```
001 /* This is the sequential version of the Mult program, used in
002  * iblOpt for demonstration purposes. It basically multiplies the
003  * Matrix A with the Matrix B and stores the result into Matrix
004  * C.
005
006  * Inputs (through defines in the Makefile):
007  *   N - matrix size
008
009  * Outputs:
010  *   the sum of all array elements
011  *   wall-clock time of the calculation phase
012
013  * Author: Michael Süß
014  */
015
016 /* needed for calculating seconds from nanoseconds */
017 #define BILLION  1000000000L;
018
019 #include <stdio.h>
020 #include <stdlib.h>
021 #include <time.h>
022
023 int A[N][N];
024 int B[N][N];
025 int C[N][N];
026
027 int main()
028 {
029   int i, j, k;
030   long sum = 0;
031   struct timespec start, stop;
032   double accTime;
033
034   /* Array initialization phase */
035   for (i = 0; i < N; i++)
036     for (j = 0; j < N; j++) {
037       A[i][j] = i;
038       B[i][j] = j;
039       C[i][j] = 0;
040     }
041
```

```
042   /* Start measuring the time */
043   clock_gettime(CLOCK_REALTIME, &start);
044
045   /* Calculation phase */
046   for (i = 0; i < N; i++) {
047     for (j = 0; j < N; j++)
048       for (k = 0; k < N; k++)
049         C[i][j] = A[i][k] * B[k][j] + C[i][j];
050   }
051
052   /* Finish time measurement */
053   clock_gettime(CLOCK_REALTIME, &stop);
054
055   accTime =
056     (stop.tv_sec - start.tv_sec) + (double) (stop.tv_nsec -
057                           start.tv_nsec)
058     / (double) BILLION;
059
060   /* Verification phase */
061   for (i = 0; i < N; i++) {
062     for (j = 0; j < N; j++)
063       sum += C[i][j];
064   }
065
066   /* print out the sum of all array elements */
067   printf("sum: %ld\n", sum);
068
069   /* print out the wall-clock time of the calculation phase */
070   printf("wall-clock time = %.2lf (N=%d, P=1)\n", accTime, N);
071
072   return EXIT_SUCCESS;
073 }
074
075
```

A.3 laplaceseq.c

```
001 /* This is the sequential version of the laplace program, used in
002  * iblOpt for demonstration purposes. It basically solves the
003  * laplace equation using a simple Jacobi - Iteration.
004
005  * Inputs (through defines in the Makefile):
```

```
006  *   N - matrix size
007
008  * Outputs:
009  *   the sum of all array elements
010  *   wall-clock time of the calculation phase
011
012  * Author: Michael Süß
013  */
014
015 /* needed for calculating seconds from nanoseconds */
016 #define BILLION  1000000000L;
017
018 #include <stdio.h>
019 #include <stdlib.h>
020 #include <time.h>
021
022 float A[N][N];
023 float B[N][N];
024
025 int main()
026 {
027   int i, j, t;
028   double sum = 0;
029   struct timespec start, stop;
030   double accTime;
031
032   /* Array initialization phase */
033   /* Initialize everything to zero */
034   for (i = 0; i < N; i++)
035     for (j = 0; j < N; j++)
036       A[i][j] = 0.;
037
038   /* except for one spike in the middle */
039   A[N / 2][N / 2] = -1.;
040
041   /* Start measuring the time */
042   clock_gettime(CLOCK_REALTIME, &start);
043
044   /* Calculation phase */
045   /* Iterate 1000 times */
046   for (t = 0; t < 1000; t++) {
047
048     /* copy new solution into old one */
```

```
049     for (i = 0; i < N; i++)
050       for (j = 0; j < N; j++)
051         B[i][j] = A[i][j];
052
053     /* calculate next iteration */
054     for (i = 1; i < N - 1; i++)
055       for (j = 1; j < N - 1; j++) {
056         A[i][j] =
057           (B[i + 1][j] + B[i - 1][j] + B[i][j + 1] +
058            B[i][j - 1]) / 4.0;
059       }
060   }
061
062   /* Finish time measurement */
063   clock_gettime(CLOCK_REALTIME, &stop);
064
065   accTime =
066     (stop.tv_sec - start.tv_sec) + (double) (stop.tv_nsec -
067                           start.tv_nsec)
068     / (double) BILLION;
069
070   /* Verification phase */
071   for (i = 0; i < N; i++) {
072     for (j = 0; j < N; j++)
073       sum += A[i][j];
074   }
075
076   /* print out the sum of all array elements */
077   printf("sum: %lf\n", sum);
078
079   /* print out the wall-clock time of the calculation phase */
080   printf("wall-clock time = %.2lf (N=%d, P=1)\n", accTime, N);
081
082   return EXIT_SUCCESS;
083 }
084
085
```

Appendix B

Parallel Program Sources

B.1 trapopara.c

```
001 /* This is the parallel version of the Trapo program, used in
002  * iblOpt for demonstration purposes. It basically transforms the
003  * Matrix B and saves the result into Matrix A.
004
005  * Inputs (through defines in the Makefile):
006  *   N - matrix size
007  *   P - number of threads (processors) to use
008
009  * Outputs:
010  *   the sum of all array elements
011  *   wall-clock time of the calculation phase
012
013  * Author: Michael Süß
014  */
015
016 /* needed for calculating seconds from nanoseconds */
017 #define BILLION  1000000000L;
018
019 #include <omp.h>
020 #include <stdio.h>
021 #include <stdlib.h>
022 #include <time.h>
023
024 int A[N][N];
025 int B[N][N];
026
027 int main()
028 {
```

```
029   int i, j, k;
030   int chunk = N / P;
031   long sum = 0;
032   struct timespec start, stop;
033   double accTime;
034
035   /* Array initialization phase */
036   for (i = 0; i < N; i++)
037     for (j = 0; j < N; j++)
038       B[i][j] = i;
039
040   omp_set_num_threads(P);
041
042   /* Start measuring the time */
043   clock_gettime(CLOCK_REALTIME, &start);
044
045   /* Timing loop, to make wall-clock-time more measurable */
046   for (k = 0; k < 1000; k++) {
047
048     /* Calculation phase */
049 #pragma omp parallel for shared(A,B) private(i,j)
schedule(static, chunk)
050     for (i = 0; i < N; i++) {
051       for (j = 0; j < N; j++)
052         A[i][j] = B[j][i];
053     }
054   }
055
056   /* Finish time measurement */
057   clock_gettime(CLOCK_REALTIME, &stop);
058
059   accTime =
060     (stop.tv_sec - start.tv_sec) + (double) (stop.tv_nsec -
061                           start.tv_nsec)
062     / (double) BILLION;
063
064   /* Verification phase */
065   for (i = 0; i < N; i++) {
066     for (j = 0; j < N; j++)
067       sum += A[i][j];
068   }
069
070   /* print out the sum of all array elements */
```

```
071   printf("sum: %ld\n", sum);
072
073   /* print out the wall-clock time of the calculation phase */
074   printf("wall-clock time = %.2lf (N=%d, P=%d)\n", accTime, N, P);
075
076   return EXIT_SUCCESS;
077 }
078
079
```

B.2 multpara.c

```
001 /* This is the parallel version of the Mult program, used in
002  * iblOpt for demonstration purposes. It basically multiplies the
003  * Matrix A with the Matrix B and stores the result into Matrix
004  * C.
005
006  * Inputs (through defines in the Makefile):
007  *   N - matrix size
008  *   P - number of threads (processors) to use
009
010  * Outputs:
011  *   the sum of all array elements
012  *   wall-clock time of the calculation phase
013
014  * Author: Michael Süß
015  */
016
017 /* needed for calculating seconds from nanoseconds */
018 #define BILLION  1000000000L;
019
020 #include <omp.h>
021 #include <stdio.h>
022 #include <stdlib.h>
023 #include <time.h>
024
025 int A[N][N];
026 int B[N][N];
027 int C[N][N];
028
029 int main()
030 {
```

```
031   int i, j, k;
032   int chunk = N / P;
033   long sum = 0;
034   struct timespec start, stop;
035   double accTime;
036
037   /* Array initialization phase */
038   for (i = 0; i < N; i++)
039     for (j = 0; j < N; j++) {
040       A[i][j] = i;
041       B[i][j] = j;
042       C[i][j] = 0;
043     }
044
045   omp_set_num_threads(P);
046
047   /* Start measuring the time */
048   clock_gettime(CLOCK_REALTIME, &start);
049
050   /* Calculation phase */
051 #pragma omp parallel for shared(A,B,C) private(i,j,k)
schedule(static, chunk)
052   for (i = 0; i < N; i++) {
053     for (j = 0; j < N; j++)
054       for (k = 0; k < N; k++)
055         C[i][j] = A[i][k] * B[k][j] + C[i][j];
056   }
057
058   /* Finish time measurement */
059   clock_gettime(CLOCK_REALTIME, &stop);
060
061   accTime =
062     (stop.tv_sec - start.tv_sec) + (double) (stop.tv_nsec -
063                           start.tv_nsec)
064     / (double) BILLION;
065
066   /* Verification phase */
067   for (i = 0; i < N; i++) {
068     for (j = 0; j < N; j++)
069       sum += C[i][j];
070   }
071
072   /* print out the sum of all array elements */
```

```
073   printf("sum: %ld\n", sum);
074
075   /* print out the wall-clock time of the calculation phase */
076   printf("wall-clock time = %.2lf (N=%d, P=%d)\n", accTime, N, P);
077
078   return EXIT_SUCCESS;
079 }
080
081
```

B.3 laplacepara.c

```
001 /* This is the parallel version of the laplace program, used in
002  * iblOpt for demonstration purposes. It basically solves the
003  * laplace equation using a simple Jacobi - Iteration.
004
005  * Inputs (through defines in the Makefile):
006  *   N - matrix size
007  *   P - number of threads (processors) to use
008
009  * Outputs:
010  *   the sum of all array elements
011  *   wall-clock time of the calculation phase
012
013  * Author: Michael Süß
014  */
015
016 /* needed for calculating seconds from nanoseconds */
017 #define BILLION  1000000000L;
018
019 #include <omp.h>
020 #include <stdio.h>
021 #include <stdlib.h>
022 #include <time.h>
023
024 float A[N][N];
025 float B[N][N];
026
027 int main()
028 {
029   int i, j, t;
030   int chunk = N / P;
```

```
031   double sum = 0;
032   struct timespec start, stop;
033   double accTime;
034
035   /* Array initialization phase */
036   /* Initialise everything to zero */
037   for (i = 0; i < N; i++)
038     for (j = 0; j < N; j++)
039       A[i][j] = 0.;
040
041   /* except for one spike in the middle */
042   A[N / 2][N / 2] = -1.;
043
044   omp_set_num_threads(P);
045
046   /* Start measuring the time */
047   clock_gettime(CLOCK_REALTIME, &start);
048
049 #pragma omp parallel private(i,j,t) shared(A,B)
050   {
051     /* Calculation phase */
052     /* Iterate 1000 times */
053     for (t = 0; t < 1000; t++) {
054
055       /* copy new solution into old one */
056 #pragma omp for schedule(static, chunk)
057       for (i = 0; i < N; i++)
058         for (j = 0; j < N; j++)
059           B[i][j] = A[i][j];
060
061       /* calculate next iteration */
062 #pragma omp for schedule(static, chunk)
063       for (i = 1; i < N - 1; i++)
064         for (j = 1; j < N - 1; j++) {
065           A[i][j] =
066             (B[i + 1][j] + B[i - 1][j] + B[i][j + 1] +
067              B[i][j - 1]) / 4.0;
068         }
069     }
070   }
071
072   /* Finish time measurement */
073   clock_gettime(CLOCK_REALTIME, &stop);
```

```
074
075   accTime =
076     (stop.tv_sec - start.tv_sec) + (double) (stop.tv_nsec -
077                           start.tv_nsec)
078     / (double) BILLION;
079
080   /* Verification phase */
081   for (i = 0; i < N; i++) {
082     for (j = 0; j < N; j++)
083       sum += A[i][j];
084   }
085
086   /* print out the sum of all array elements */
087   printf("sum: %lf\n", sum);
088
089   /* print out the wall-clock time of the calculation phase */
090   printf("wall-clock time = %.2lf (N=%d, P=%d)\n", accTime, N, P);
091
092   return EXIT_SUCCESS;
093 }
094
095
```

Bibliography

[AL97] Emile H.L. Aarts and Jan K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computer capabilities. In *Proceedings AFIPS Spring Joint Computer Conference*, volume 30, pages 483–485, 1967.

[BS96] Lou Baker and Bradley J. Smith. *Parallel Programming*. The McGraw-Hill Companies, 1996.

[CDK00] Rohit Chandra, Leonardo Dagum, and Dave Kohr. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000.

[DRV00] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser, 2000.

[Fly66] Michael J. Flynn. Very high speed computing systems. In *Proceedings IEEE*, volume 54, pages 1901–1909, 1966.

[Gei94] Al Geist. *Pvm: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[GNL98] William Gropp, Bill Nitzberg, and Ewing Lusk. *Mpi: The Complete Reference*. MIT Press, 1998.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, 3. edition, 2003.

[HPC] HPCinfo: Hardware — SMP. `http://www.epcc.ed.ac.uk/HPCinfo/hware-smp.html`.

[KELS62] Tom Kilburn, Dai B.G. Edwards, M.J. Lanigan, and Frank H. Sumner. One-level storage system. *IRE Trans. on Electronic Computers*, pages 223–235, April 1962.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison Wesley Longman, 1999.

[KRC97] Mahmut T. Kandemir, Jagannathan Ramanujam, and Alok Choudhary. Compiler algorithms for optimizing locality and parallelism on shared and distributed memory machines. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation techniques*, pages 236–247, 1997.

[KS95] Dattatraya Kulkarni and Michael Stumm. Linear loop transformations in optimizing compilers for parallel machines. *The Australian Computer Journal*, 27(2):41–50, 1995.

[Lab03] Lawrence Livermore National Laboratory. OpenMP. `http://www.llnl.gov/computing/tutorials/openMP/`, 2003.

[Lap03] Laplace's equation. `http://en.wikipedia.org/wiki/Laplace's_equation`, September 2003. Published in Wikipedia, The Free Encyclopedia.

[Leo98] Claudia Leopold. Arranging statements and data of program instances for locality. *Future Generation Computer Systems*, 14(5–6):293–311, 1998.

[Leo01a] Claudia Leopold. *Parallel And Distributed Computing*. John Wiley & Sons, 2001.

[Leo01b] Claudia Leopold. Structuring statement sequences in instance-based locality optimization. *Future Generation Computer Systems*, 17(4):425–440, 2001.

[Leo01c] Claudia Leopold. A tool for instance-based locality optimization. unpublished manuscript, 2001.

[LL98] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24:445–475, 1998.

[PH98] David A. Patterson and John L. Hennessy. *Computer Organisation and Design: The Hardware / Software Interface*. Morgan Kaufmann Publishers, 2. edition, 1998.

[RR00] Thomas Rauber and Gundula Ruenger. *Parallele und verteilte Programmierung*. Springer, 2000.

[Sch03] Andreas Schönfeld. Hessischer Hochleistungsrechner–HHLR. `http://www.tu-darmstadt.de/hrz/hhlr/`, September 2003.

[SM97] Sharad K. Singhai and Kathryn S. McKinley. A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, pages 340–355, 1997.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2. edition, 2001.

[TG99] Andrew S. Tanenbaum and James Goodman. *Structured Computer Organisation*. Prentice Hall, 4. edition, 1999.

[WA99] Barry Wilkinson and Michael Allen. *Parallel Programming*. Prentice Hall, 1999.

[Wol96] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

[Zei96] Eberhard Zeidler. *Teubner-Taschenbuch der Mathematik*. B.G. Teubner Verlagsgesellschaft Leipzig, 1996.

Index

Amdahl's Law, 13

bit
    dirty, 10
    valid, 10
block, 6

cache, 8
    coherence, 18, 36
    data, 8
    direct mapped, 8
    directory based, 19
    fully associative, 8, 36
    hit time, 31, 36
    instruction, 8
    line, 8
      size, 31
    misses, *see* miss(es)
    no–write–allocate, 10
    replacement
      least–recently used, 10, 36
      random, 10
    set associative, 8
    size, 31
    snooping, 19
    sync time, 31, 34, 36
    write update, 19
    write–allocate, 10, 36
    write–back, 10, 36
    write–broadcast, 19
    write–invalidate, 19, 36
    write–through, 10, 53
    write–update, 53
CHT, *see* cache, hit time
Clusters, 17
coherence, *see* cache, coherence
communication
    message passing, 20
    shared memory, 20
CST, *see* cache, sync time

dag, *see* directed acyclic graph
data instance, 26, 28
dependency acyclic graph, 44
dependency analysis, 24
DI, *see* data instance
directed acyclic graph, 29, 41
dirty bit, *see* bit, dirty

efficiency, 13, 46, 49, 51

generalization phase, *see* phase, generalization
Grids, 17

Harvard Architecture, 8
Hessischer Hochleistungsrechner, 46
HHLR, *see* Hessischer Hochleistungsrechner
hit(s), 7
    rate, 7
    ratio, 7

iblOpt, 2, 24–26, 36, 43, 52
implementation, 40–42
instantiation phase, *see* phase, instantiation

Jacobi Iteration, 38

laplace equation, 38, 49
local search, 27
locality, 5, 8, 24, 44, 53
    code, 6
    data, 6

sequential, 6
spatial, 6
temporal, 5
loop tiling, 45, 51

matrix
multiplication, 38, 46
transposition, 38, 43
memory
access–time, *see* time, memory–access
load time, *see* time, memory–load
main, 7
virtual, 7
message passing communication, 20
Message Passing Interface, 20
method
semi–automatic, *see* semi–automatic method
middleware, 20
MIMD, *see* Multiple Instruction Multiple Data
MISD, *see* Multiple Instruction Single Data
miss(es), 7
capacity, 9
cold–start, 9
collision, 9
compulsory, 9, 47
conflict, 9
penalty, 7
rate, 7
MLT, *see* time, memory–load
MPI, *see* Message Passing Interface
Multiple Address Spaces, 16
Multiple Instruction Multiple Data, 15
Multiple Instruction Single Data, 14

neighbor, 27
neighborhood, 27
No Remote Memory Access, 17
Nonuniform Memory Access, 16, 53
NORMA, *see* No Remote Memory Access
NUMA, *see* Nonuniform Memory Access

objective function, 27, 33–35, 53
OpenMP, 20–23
master thread, 21
parallel region, 21
optimization algorithm, 28–33
optimization phase, *see* phase, optimization

page(s), 7
fault, 7
parallel
efficiency, *see* efficiency
speedup, *see* speedup
Parallel Virtual Machine, 20
parallelization
automatic, 23–25
phase
generalization, 25, 45
instantiation, 25, 43
optimization, 25, 44
PI, *see* program instance
program instance, 2, 26, 28, 29, 43
PVM, *see* Parallel Virtual Machine

RAM, *see* Random Access Memory
Random Access Memory, 7
registers, 7
regularity optimization, 26, 53
runtime simulator, 35–36, 52, 53

semi–automatic method, 2, 24–26, 45
shared memory communication, 20
SI, *see* statement instance, 28
SIMD, *see* Single Instruction Multiple Data
Single Address Space, 15
Single Instruction Multiple Data, 14, 54
Single Instruction Single Data, 14
SISD, *see* Single Instruction Single Data
SMP, *see* Symmetric Multiprocessor
speedup, 12, 46
superlinear, 13, 46
statement instance, 2, 26, 28
stencil codes, 39

strip mining, 46
Symmetric Multiprocessor, 2, 15, 53

threads, 20
    Java threads, 20
    pthreads, 20
tiling, *see* loop tiling
time
    elapsed, 11
    execution, 11
    memory–access, 33
      accumulated, 34
    memory–load, 31, 36
    response, 11
    wall–clock, 11, 33, 46, 47, 51

UMA, *see* Uniform Memory Access
Uniform Memory Access, 15

valid bit, *see* bit, valid
variance, 35

wall–clock–time, *see* time, wall–clock

Statement

Erklärung

Hereby I reassure having written the presented work independently and by using only the listed sources and facilities.

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, October 30, 2003

Michael Süß