A Brief Introduction to Parallel Computing*

Marcin Paprzycki† Przemysław Stpiczyński‡

August 2, 2006

Abstract

In this chapter a concise overview of the fundamentals of parallel computing is presented. It is intended for readers who are familiar with the general aspects of computing but are new to high performance and parallel computing. This chapter provides a general overview of the field and a requisite background for the issues considered in subsequent chapters.

Contents

1 Introduction **1**

2 Why parallel computing ? **2**

3 Architectures **3**

3.1 Inside of a processor 4
3.2 Memory hierarchies 6
3.3 Flynn's Taxonomy 7
3.4 SIMD computers 7
3.5 MIMD computers 8
3.5.1 Shared memory computers 9
3.5.2 Distributed memory computers 10
3.6 Architecture convergence 10
3.7 Clusters 10
3.8 The Grid 11

*The first chapter of the book *Parallel Computing and Statistics.*

†Computer Science Department, Oklahoma State University, Tulsa, OK 74106, USA, e-mail: marcin@cs.okstate.edu

‡Department of Computer Science, Marie Curie-Sklodowska University, Lublin, Poland, e-mail: przem@hektor.umcs.lublin.pl

4 Levels of parallelization **12**
4.1 Inside of a single processor 12
4.2 Shared memory parallelism 13
4.3 Distributed memory parallelism 14
4.4 Grid parallelization 15

5 Theoretical models for performance analysis **16**
5.1 Basic raw performance 16
5.2 Hockney's and Jesshope's model for vector processing 17
5.3 Amdahl's law 18

6 Programming parallel computers **23**
6.1 Optimizing compilers 24
6.2 Language based parallelization 26
6.3 Shared memory parallel algorithms and OpenMP 27
6.4 Distributed memory parallelization 29
6.4.1 PVM and MPI 30
6.5 Shared-distributed memory environments 33
6.6 Parallelization of existing codes 34
6.7 Library-based parallelization 36
6.7.1 BLAS and LAPACK 36
6.7.2 BLACS, PBLAS and ScaLAPACK 41

7 Concluding Remarks **44**

1 Introduction

Over the past twenty years, parallel computing has emerged from the enclaves of research institutions and cutting edge technology firms and entered the mainstream. While multi-processor machines were once a marvel of technology, today most personal computers arrive prebuilt with multiple processing units (i.e. main processor/CPU, video processor, sound processor, etc.), with computational power and memory substantially larger than those of the top of the line, high-performance workstations of just a few years ago. More importantly, the increased availability of inexpensive components has made "desktop" systems with two and four processors available for the price ranging from $5,000 to $15,000, while the proliferation of eight and sixteen processor systems has been delayed only by the recent economic downturn.

In this chapter, a concise overview of the fundamentals of parallel computing is presented. It is intended for readers who are familiar with the general aspects of computing but are new to high performance and parallel computing and provides a general overview of the field and a background for the issues considered in following chapters. Specialists in high performance and/or parallel computing may wish to skip directly to the subsequent material.

2 Why parallel computing ?

Why parallel computing? The answer is simple: because there exists a need to solve large problems – the kinds that often arise, among others, in the application of statistical methods to large data sets.

While the ongoing applicability of Moore's Law promises uninterrupted increase in (single) processor "power," the size of the problems that researchers are interested in solving is increasing even faster. Parallel computing becomes a viable bridge across this gap when it is possible to harness multiple processors in a straightforward way, allowing non-specialists to take advantage of available computational power. This does not imply that the procedures involved are trivial (if this were the case, volumes like the present one would not be needed), but merely that the economics of computing is relatively simple. If by combining four processors it is possible to cut the execution time of a large program by half, this is most likely worth the effort. Of interest also is the trend in the overall growth of the world's most powerful computers, which follow the pattern described by Moore's Law (doubling approximately every 18 months) [35]. Accordingly, the size of problems that can be feasibly solved also nearly doubles every 18 months. However, to be able to efficiently solve problems on machines with hundreds and thousands of processors, knowledge must be accumulated and experience built across the spectrum of available architectures.

Large problems from many areas of scientific endeavors are often used to illustrate the need for large-scale parallel computing. Some examples include (this list is certainly not exhaustive, but rather is presented here to indicate the breadth of the need for parallel computing):

- Earth environment prediction,
- nuclear weapons testing (the ASCI program),
- quantum chemistry,

- computational biology,
- data mining for large and very large datasets,
- astronomy and cosmology,
- cryptography,
- approximate algorithms for $\mathcal{NP}$-complete problems.

The first two problems on the list led to the creation of the two largest supercomputers in the world, which achieve 35.86 and 7.72 TFlops (10×10^{12} floating point operations per second), respectively. Unfortunately, their performance still does not reach the level necessary to accomplish the original scientific goals set forth by the researchers. For instance, the estimated necessary sustained speed of computations for the ASCI program is of the order of 10^{15} floating point operations per second. Obviously, even with the exponential growth of computational power promised by Moore's Law, a single processor will not be able to achieve this level of performance any time soon (if ever); therefore combining multiple computational units is the only possible solution.

In summary, parallel computing and the construction of large multiprocessor systems has come to the forefront because of the need to solve large computationally intensive problems. While it is relatively easy to combine a few (2-16) processors to work on a problem on a small scale (it is in fact much easier and cheaper than to acquire a single processor that would be as powerful on its own), multiprocessing is the **only way** to reach the necessary performance on a large scale.

3 Architectures

In this section, some important issues in computer architectures are briefly described (a more detailed treatment of hardware issues can be found in Chapter **??**). For the purposes of this book, parallel computing is not considered a goal in its own right but rather a means of solving large computationally intensive problems. This being the case, performance needs to be considered on all "levels" of a multiprocessor system, to assure that its overall efficiency is optimal. Starting from the processor itself, the following issues need to be addressed in order to indicate how its performance can be maximized: (1) what is happening inside of a processor, (2) how can data be efficiently supplied to fast processors.

3.1 Inside of a processor

One of the important features that drive the performance of modern processors is *pipelining*, whose origins can be traced back to the CDC 6600 computer, in which it was for the first time successfully applied in a commercial processor. Pipelining is based on a very simple observation: in a classical von Neumann architecture, processors complete consecutive operations in a fetch-execute cycle, e.g. each instruction is fetched, decoded, the necessary data is fetched, the instruction is executed, the results are stored and only then is the processing of the next instruction initiated. This means that a single instruction is executed in five steps and, after the start-up period, each instruction is completed every cycle (see Figure 1). Assuming that there is no direct dependency between consecutive instructions, while one instruction is being decoded, another may be fetched. So, when the data for the first instruction is fetched (third step), yet another instruction is introduced. In this way, after a start-up period of 5 cycles, the pipeline fills in (5 instructions are in various stages of execution) and an instruction is "completed" every step (instead of every 5 steps). In the case when there is a direct dependency between consecutive operations, the compiler can often rearrange the instructions in such a way as to keep these operations separated in order to maintain independence and prevent the disruption of the pipeline. Finally, when branch instructions are encountered, an educated guess is made (and there exists a large body of research about how to make a correct branch prediction) and "the most likely" of the available branches is selected. In modern architectures, the success rate of branch prediction is above 90%.

This general idea of pipelining has been further extended in the following directions. First, it was observed that scientific computations very often involve vectors and matrices. Accordingly, special pipelines have been developed for floating point vector operations (see Figure 1). In addition, the multiplication of a vector by a constant is often followed by vector addition: operations of the type $\alpha x + y$ (where x and y are vectors). For this class of operations, it is possible to further improve processor performance by sending the results from the multiplication pipeline directly into the addition pipeline, without storing the intermediate results. This process is called *chaining*. Second, operations on integers differ from operations on floating point numbers in the amount of steps required for their completion. For instance, when two integers are added, since both of them are objects belonging to the same range and represented as two strings of 32 bits, the only operation to be performed is actual addition. In the case of floating point

	Cycle					
Stage	**1**	**2**	**3**	**4**	**5**	**6**
fetch operands	A[1], B[1]	A[2], B[2]	A[3], B[3]	A[4], B[4]	A[5], B[5]	A[6], B[6]
adjust exponents		A[1], B[1]	A[2], B[2]	A[3], B[3]	A[4], B[4]	A[5], B[5]
execute multiplication			A[1]*B[1]	A[2]*B[2]	A[3]*B[3]	A[4]*B[4]
normalize result				A[1]*B[1]	A[2]*B[2]	A[3]*B[3]
store result					A[1]*B[1]	A[2]*B[2]

Figure 1: Partial representation of pipelined floating-point multiplication of elements of a vector

addition, the two have to be assumed to belong to different ranges of available numbers and have to be aligned to be added. This involves operations on both their mantissas and exponents. After the operation is completed, the result needs to be "rescaled" to the desired final representation, which involves further operations on the mantissa and the exponent of the result (see Figure 1). This difference opens up the possibility of developing processors with separate pipelines for integer and floating point operations. Finally, since it is often possible to reorganize the instruction stream to sufficiently separate dependent operations and since many operations on long vectors can be divided into independent operations on sub-vectors (e.g. vector addition, vector scaling etc.), it is also possible to introduce multiple integer and floating point pipelines. Currently, all modern processors – including products from Intel and AMD – are built this way.

While the processors are constantly increasing their speed, an important problem arises: how to provide them with data sufficiently fast. The memory subsystem is typically either too slow to service the processor or the implementation of a memory service that is fast enough is too expensive to be economically feasible, which results in a bottleneck in the processing capabilities of the computer. It becomes impossible to feed the ever-faster processors with data at an appropriately fast rate. Attempts to minimize the effects of this bottleneck have led to the development of hierarchical memory.

3.2 Memory hierarchies

There are two ways of addressing the limitations of the memory subsystem. One way is to consider only the performance of the memory and to install

the fastest subsystem available at a time. To further increase its speed, the memory can be divided into separately accessible and refreshable memory "banks." Given this setup, if the data is laid out and accessed in an optimal way, each consecutive element is retrieved from a separate memory bank and this memory bank is ready before the next access is requested. Unfortunately, if consecutive elements are retrieved from the same memory bank, a memory bank conflict occurs (the next element cannot be retrieved until the memory bank is refreshed). This can result in the reduction of performance even by a factor of seven [78, 80, 79, 42]. This approach to the memory bottleneck problem was a staple of Cray Research and their architectures e.g. Cray Y-MP, which while characterized by very high memory throughput, was chiefly responsible for the prices of the order of $ 20 million (in 1990, with an academic discount) for a complete system.

Extremely fast uniform-speed memory cannot be afforded in quantity except by a few power users, e.g. very large companies, the government, its military and their research laboratories, etc. Thus the need for an alternative approach – cache memory. Here, a relatively small, high-grade and fast memory subsystem is inserted between the processor and the main memory, which is of a lower grade but proportionally larger. Cache memory stores both the instructions that are to be utilized by the processor and the data, which is predicted to be used next. Since cache memory is faster than the main memory, it can increase the data throughput (assuming that it contains the correct data that is to be used in subsequent steps). Modern systems typically have at least two levels of cache memory and the general rule that characterizes such systems is: the further "away" from the processor, the larger and slower the available memory and the longer it takes for the data to reach the processor. There are downsides to this approach, of course. Since each level of cache has a different speed for every level of cache in the system, different data latencies are introduced. To utilize the hierarchical memory system to its fullest extent, algorithms have to be oriented toward *data locality*. First, groups of data elements that are to be worked on together should be moved up the memory hierarchy (closer to the processor) together. Second, **all** necessary operations on these data elements should be performed while they are stored in the closest cache to the processor and the algorithm should not return to these elements in subsequent steps. For example, in the context of numerical linear algebra, this is achieved through the application of block-oriented algorithms based on the level 3 BLAS kernels (see section 6.7.1 for more details) and careful selection of blocksizes (utilizing, for instance, tools developed by the ATLAS project [99]).

3.3 Flynn's Taxonomy

Now that some of the techniques used to optimize the performance of single processor systems have been presented, a look at possible configurations for multiple processors is in order (for more detailed treatment of these topics, see Chapter **??**). The old but still useful general computer taxonomy introduced by Flynn in 1966 [50] provides a good starting point. By considering the fact that information processing, which takes place inside of a computer, can be conceptualized in terms of interactions between data streams and instructions streams, Flynn was able to classify computer architectures into four types:

- SISD – single instruction stream/single data stream, which includes most of the von Neumann type computers,
- MISD – multiple instruction streams/single data stream, which describes various special computers but no particular class of machines,
- SIMD – single instruction stream/multiple data streams, the architecture of parallel processor "arrays," and
- MIMD – multiple instruction streams/multiple data streams, which includes most modern parallel computers.

The last two types of machines, SIMD and MIMD, are at the center of interest in this book since most of the currently used parallel systems fall into one of the two categories.

3.4 SIMD computers

SIMD-based computers are characterized by a relatively large number of relatively weak processors, each associated with a relatively small memory. These processors are combined into a matrix-like topology, hence the popular name of this category: "processor arrays". This computational matrix is connected to a controller unit (usually a top of the line workstation), where program compilation and array processing management takes place (see Figure 2). For program execution, each processor performs operations on separate data streams; all of the processors may perform the same operation, or some of them may skip a given operation or a sequence of operations. In the past, the primary vendors producing such machines were ICL, MasPar and Thinking Machines. Currently, this class of machines is no longer produced for the mainstream of parallel computing. One of the main advantages of

SIMD computers was the fact that the processors work synchronously, which enables relatively easy program tracing and debugging. Unfortunately, this advantage comes at a price. In experiments with actual SIMD computers, users realized that it is relatively difficult to use them for unstructured problems and, in general, all problems that require a high level of flexibility of data manipulation and data transfer between processing units. Currently, most of these machines have disappeared from the market, although there are researchers, who claim that SIMD machines are about to make a comeback [87].

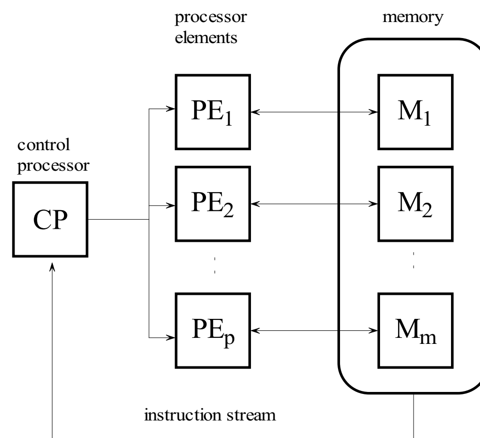

Figure 2: SIMD multiprocessor

3.5 MIMD computers

The class of MIMD-based parallel computers can be divided into two subcategories: shared memory and distributed memory systems. This division is based on the way that the memory and the processors are connected (see Figure 3).

3.5.1 Shared memory computers

Computers of this type consist of a number of processors that are connected to the main (global) memory. The memory connection is facilitated by fast

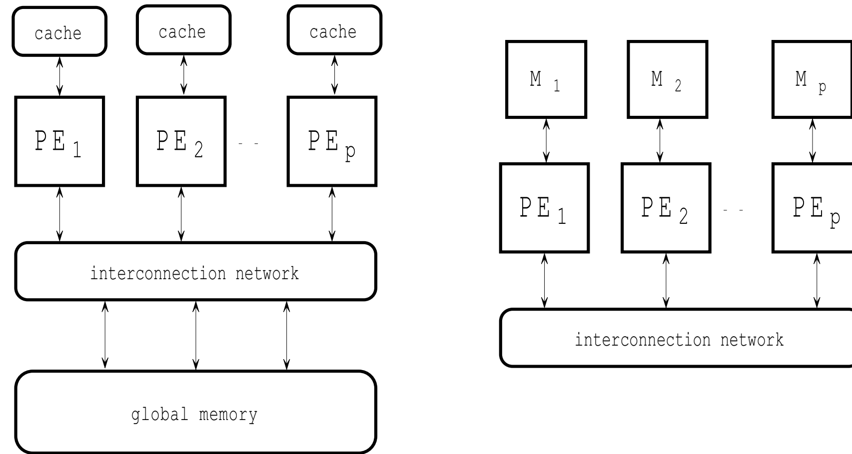

Figure 3: Shared and distributed memory multiprocessors

bus technology or a variety of switch types (e.g. omega, butterfly, etc.) These machines were initially developed with practically memory-less processors but were later equipped with cache memory, which resulted in relatively complicated data management / cache coherency issues. Many companies have produced shared memory computers over the years, including: BBN, Alliant, Sequent, Convex, Cray and SGI. While these systems were relatively easy to program (with loop parallelization being the simplest and the most efficient means of achieving parallelism), computational practice exposed some important hardware limitations in their design. The most important of them is the fact that it always was and still is almost impossible to scale shared memory architectures to more than 32 processors and to simultaneously avoid saturating the bandwidth between the processors and the global memory. Nevertheless, one can observe a resurgence of shared memory computers in the form of: multiprocessor desktops, which usually contain two or four processors; midrange systems with four or eight processors, primarily used as servers; and high performance nodes for the top of the line parallel computers, which usually contain 16, 32 or even 64 processors

(in the latter case special switching technology is applied).

3.5.2 Distributed memory computers

Since the early days of parallel computing, the second most popular architecture, is distributed memory technology, supported over time among others by: Intel, NCube, Inmos, Convex, Cray and IBM. In this configuration, processing units are composed of a processor and a large local memory and are interconnected through a structured network. There is no global memory. Typical topologies for these structured networks are meshes, toruses and hypercubes. While code for distributed memory computers is relatively more difficult to write and debug, the architecture can be scaled to a large number of processors. Even some of the early distributed memory machines successfully employed up to 1024 processors.

3.6 Architecture convergence

Over time the experiences of early adopters led to an evolution in the design of parallel computers architectures, a process which became even more accelerated with the end of the Cold War and the substantial drop in military funding for high performance computing, which resulted in bankruptcies or product line changes (moving away from producing parallel computer hardware) for a number of parallel computer vendors (among others, Kendall Square Research, Alliant, BBN and Thinking Machines). SIMD machines have been all but eliminated, although many of their conceptual and technological advances have been incorporated into modern processor design (video processors in particular). The development of shared memory machines had almost become dormant, until the rebirth of the architecture in desktop PCs and servers. Distributed memory architectures have been by far the most successful. This success has been further substantiated in 1995, when NASA scientists successfully completed the Beowulf project, which paved the way toward cluster computing [4].

3.7 Clusters

While the largest computers in the world are still custom built for the highest performance, they cost tens of millions of dollars as well. Clusters made high performance parallel computing available to those with much smaller budgets. The idea is to combine commodity-off-the-shelf (COTS) components to create a parallel computer. The Beowulf project was the forerunner in this approach.

It is now possible to network 16 top of the line PCs using a fast switch to form a parallel computer that is substantially more powerful than the fastest machine in the world in 1990 for no more than $50,000 (a 400 fold price decrease). What is even more interesting is the possibility of combining higher-end shared memory PCs and servers into clusters. One can still use the same 16-port switch but with each node having four processors, for example, a resulting parallel computer has 64 processors. Notice however, that in the computational practice, it is the connectivity between the cluster nodes that is the weakest point of many clusters. Low throughput switches can result in imbalanced systems and become a major performance bottleneck, especially when more powerful nodes are used in the cluster. Regardless of possible drawbacks and limitations, due to their excellent price-performance ratio, clusters have been successfully applied in the industry to solve practical problems (see for instance [64]).

It is important to stress here that clusters are **not** a different architectural category of parallel computers in Flynn's taxonomy; they are merely a practical way of building efficient low cost distributed memory MIMD computers.

3.8 The Grid

The most recent developments in parallel computing have been in the area of *grid computing*. The idea is based on the metaphor of the electric power grid. If one plugs-in an appliance to the electric grid, one does not care about where the electricity comes from (as long as it is available). In the same fashion, one could plug a computer into the computer grid, request that a computational task be completed, and not have to care about where the actual computation takes place [51]. It can be said that the first successful grid-like computing endeavor was the SETI@home research project, in which the computers of more than 4,400,000 users work "together" in sifting through data acquired by the Arecibo radio telescope, searching for extraterrestrials. At the time of this writing, the SETI@home network of machines delivered approximately $54.7 \cdot 10^{12}$ floating point operations per second. Most of this power is gained by utilizing the **unused otherwise** cycles on users' computers. In the same way, computers connected to a grid will be able to combine their computational power / unused cycles to serve their customers. While a very large amount of research still remains to be completed before the GRID becomes a reality outside special academic and industrial projects, this is one of the most interesting and most fashionable areas in parallel computing today. This latter fact is signified by a

flurry of publications devoted to grid computing that appear outside of the realm of peer reviewed journals and conference proceedings (see for instance [24, 25, 67, 72, 93]). As an interesting footnote illustrating the interest in this approach, it can be noted that the PC maker Gateway tries to use its in-store demo computers as a computational grid and plans on making profit in this way to support company's bottom line [88, 75].

Summarizing, most of modern computer architectures used to solve large computationally intensive problems involve multiple levels of parallel computing: from the processors with multiple integer and floating point pipelines and special vector processing units through multiple processors combined into shared memory computers followed by multiple processing units tightly integrated into distributed memory parallel computers to multiple computers (single processor and parallel) working together in a computational grid.

4 Levels of parallelization

It will be useful to consider as an example a modern parallel computer that consists of a number of state of the art processors (vector, RISC, x86, IA-64, etc.). These processors are combined into shared memory modules. The machine consists of a number of such modules combined into distributed memory parallel computer (possibly a cluster). The computer needs to be examined from the "inside out" – what is happening inside a single processor, what is happening inside a single module, and what is happening in the whole machine. Finally, it will be assumed that a number of computers of this type are combined together into a loosely-connected network. For the purposes of this discussion, all issues related to memory banks, cache memory and memory hierarchies are omitted. They have been addressed previously and while they are crucial to the overall performance of the system they do not affect the available parallelism directly.

4.1 Inside of a single processor

As mentioned above (Section 3.1), modern day desktop processors such as those developed by Intel, AMD, IBM, etc., are already highly parallelized. Such processors have multiple pipelines for integer and floating point operations [9], so two different levels of parallelization can be considered. First, the depth of the pipeline: if a pipeline of depth k is used then k operations can be executed at the same time. Second, number of pipelines: assuming that there are l integer pipelines of depth k_1 and m floating point pipelines

of depth k_2 in a given processor, and that all pipelines are operating at maximum capacity at a given moment, then $k_1 \cdot l + k_2 \cdot m$ operations are executed by the processor concurrently in every cycle.

Availability of this level of parallelism, which is often called *microparallelism*, is a function of dependencies inside a stream of machine language operations. These dependencies are analyzed and microparallelism is supported: by the logic unit inside of the processor on the hardware level and by the compiler and compiler supplied optimization on the software level. Typically, the user does not have to and does not need to have any control over microparallelization as it should be furnished automatically by the system.

4.2 Shared memory parallelism

While multiple operations are executed in parallel inside each processor at every step of program execution, parallelism occurring at higher levels is of real interest. In the case of an architecture, in which multiple processors are connected to a global (logically and physically) shared memory, the *typical* way of introducing parallelization is to perform similar operations on subsets of data. The most natural algorithmic level of achieving such parallelization is by dividing between multiple processors work performed by a loop. In particular, a given loop (simple or the outermost loop of a nested loop structure) is divided into as many parts as there are processors (while this is not required and a loop could be divided into a different number of parts, assuming that the number of parts equals the number of processors, is most natural and can be done without loss of generality). Subsequently, each part is executed independently by a separate processor. This kind of parallelization is often called *medium-grain* parallelization and is supported either through a set of special directives (see section 6.3 below) or through high-level language extensions (see chapter **??**).

A large body of research has been and continues to be devoted to developing compilers capable of automatically generating this level of parallelization. Unfortunately, the results have been disappointing thus far. Parallelizing compilers are relatively successful in generating parallel code with simple loops, addition of two vectors, matrix multiplication, etc. However, in more complicated cases, e.g. when functions are called inside of loops, the code must still be manually divided into parallel units.

4.3 Distributed memory parallelism

The most typical approach to distributed memory parallelization is to create independent programming units that will execute separate work units (which may or may not be similar to each other), with these units communicating with each other via message passing. This latter functionality, although necessary, is expensive (in terms of the time required for messages to reach their destination and thus in terms of the latency introduced to the system). Accordingly, minimizing the number of messages passed between components becomes an important goal of program design. One must seek to divide a distributed parallel program into large computational units that are as independent from each other as possible and only rarely communicate. While it is possible that each work unit is completely different from others (e.g. some of them advance the solution of a differential equation, others perform interpolation, while others still use the result of interpolation to generate on-the-fly visualization of the solution), this is rarely the case. Most often each work unit is a derivative of the main program and performs the same subset of operations as the other work units but on separate data sets. This type of an approach is called $SPMD$ (single program, multiple data) and often referred to as *coarse level parallelization*. Obviously, this level of parallelization must be implemented manually by the programmer as the division of work is based on a *semantic* analysis of the algorithm(s) used to solve the problem (and possibly their interdependencies). One of the more important problems for distributed memory parallelization is also the question of load balancing. Since each computational unit is relatively independent (which is especially the case when they perform completely different functions), it may require different time to complete its work. This may, in turn, lead to a situation when all processors but one remain idle as they wait for the last one to complete its job. (A theoretical model, which analyzes and illustrates the negative effects of such a situation, can be found in Section 5.3.) The last example illustrates the fact that while the work units should be large and independent, they should also complete their tasks within a similar time, which makes the SPMD approach somewhat more attractive than the division of work into completely independent units.

It should be noted that although these are the principle methods of shared and distributed memory parallelization, there are alternative ways as well. The approaches described above try to make the software “match” the underlying hardware. However, it is also possible to treat a shared memory machine as a distributed memory computer and apply approaches based on message passing. Although message passing does introduce certain overhead,

this approach often results in good performance if implemented efficiently [83, 82]. The converse approach is to treat a distributed memory computer as a shared memory system and rely on the mechanisms provided by the vendor or third party software to emulate logically shared memory implemented on physically distributed memory hardware. Unfortunately, such an approach is impractical for real applications: not a single existing shared memory emulator is efficient enough to support "virtual" shared memory in a production environment. Finally, one needs to consider the hybrid hardware, where shared memory nodes have been combined into a distributed memory configuration, which results in a distributed shared memory computer. Here, again, treating such a machine as a distributed memory computer and applying appropriate parallelization techniques is usually more successful than treating it as a shared memory environment. Nevertheless, it is the combined approach of distributed and shared memory techniques that can be expected to be the most efficient in using the underlying architecture.

4.4 Grid parallelization

In grid computing, a number of computers (irrespective of their individual architectures) are loosely connected via a network. In the most general case, each machine (including the properties of connections between them) is assumed to be different. This makes for an extremely heterogeneous system, which requires the coarsest level of parallelization since the work must be divided into independent units that can be completed on different computers at different speed and returned to the main solution coordinator at any time and in any order, without compromising the integrity of the solution. Although there are tasks that are naturally amenable to this level of parallelization, a broader applicability of this approach requires much further research and infrastructure development. Examples of successfully tested tasks include the analysis of very large sets of independent data blocks, in which the problem lies in the total size of data to be analyzed (such as in the SETI@Home project [48]).

In summary, there are multiple levels at which parallelization can occur. The simplest *microparallelization* takes place inside a single processor and usually does not require the intervention of the programmer to implement. *Medium-grain* parallelization is associated with language supported and/or loop level parallelization. While some headway has been made in automating this level of parallelization with optimizing compilers, the results of these attempts are only moderately satisfactory. *Coarse-grain* parallelization is asso-

ciated with distributed memory parallel computers and is almost exclusively introduced by the programmer. Finally, grid-level parallelization is currently the focus of intensive research and, while it is a very promising model for solving large problems, its applicability in the foreseeable future will probably continue to be limited to certain classes of computational problems, viz. those that belong to the "large scale embarrassingly parallel" category.

5 Theoretical models for performance analysis

After looking at the basic issues in the design of parallel systems, the theoretical analysis of parallel algorithms executed on these systems will be considered. There are a number of models that can be employed to predict the parallel usability of a given algorithm. While the approach advocated here is not uncontroversial, it could be argued that, for a user in the context of scientific computing, the most important criterion defining the usability of a given approach is its execution time (the speed of execution of an algorithm on a given machine). Obviously, this assumes that a particular algorithm, after parallelization, has to correctly solve the problem (under appropriate conditions of correctness).

Some further assumptions need to be made explicit. Since this book is concerned with computational statistics, it can be assumed that floating-point operations are the most important (time consuming) operations that will be performed during algorithm execution. It can also be assumed that data most often will be stored/represented as vectors and matrices. In this context, a parallel algorithm can be characterized by a number of parameters that influence its performance, e.g. size of vectors and matrices, data layout, number of processors of the parallel computer, etc. Algorithm analysis with a particular computational model in mind will allow initial optimization of these parameters without requiring testing on a real machine or at least a preliminary assessment of possible algorithm efficiency.

5.1 Basic raw performance

The processing speed of computers involved in scientific calculations is usually expressed in terms of a number of floating point operations completed per second, a measure used above to describe the computational power of the world's largest supercomputers. For a long time, the basic measure was Mflops expressed as:

$$r = \frac{N}{t} \text{ Mflops}, \tag{1}$$

where N represents a number of floating point operations executed in t microseconds. Obviously, when N floating point operations is executed with an average speed of r Mflops, the execution time of a given algorithm can be expressed as:

$$t = \frac{N}{r}. \quad (2)$$

Due to the geometric increase in available speeds of computer hardware, the Mflop measure has been superseded by higher-order measures: Gflops (gigaflops), Tflops (teraflops) and even Pflops (petaflops) = 10^{15} floating point operations per second. The floating point operations rate can be used to characterize an algorithm executing on a given machine independently of the particular characteristics of the hardware, on which the algorithm is executed, as well as to describe the hardware itself. Many vendors of parallel computers advertise a theoretical peak performance for their machines; this is the maximum speed with which any algorithm can be potentially executed on their hardware. Of course, in computational practice (outside of special simplified cases, such as matrix multiplication), this performance is unattainable. At the same time, however, it indicates what performance can potentially be expected from a given machine.

5.2 Hockney's and Jesshope's model for vector processing

Although this chapter is focused on parallel computers and their performance, it is useful to start with a model for vector processors. While this model is relatively dated, its applicability has recently been revived with the development of the Earth Simulator, the largest supercomputer in the world, which was built by the NEC Corporation on a foundation of proprietary vector processors. In addition, since most modern processors consist of multiple pipelines, performance of each such pipeline can be also conceptualized in terms of the vector performance model presented here. For vector computers, the performance r_N of a vector-processing loop of length N can be expressed in terms of two parameters r_∞ and $n_{1/2}$, which are specific for a given type of a loop and a vector computer [65]. The first parameter represents the performance in MFlops for a very long loop, while the second the loop length for which a performance of about $r_\infty/2$ is achieved. Then

$$r_N = \frac{r_\infty}{n_{1/2}/N + 1} \text{ MFlops.} \quad (3)$$

This model can be applied to predict the execution time of different loops.

For example, the vector update in the form

$$x \leftarrow x + \alpha y$$

(i.e. the `_AXPY` operation) can be expressed as a loop of the length N in which each repetition consists of two floating-point operations. Thus, the execution time of `_AXPY` is

$$T_{AXPY}(N) = \frac{2N}{10^6 r_N} = \frac{2 \cdot 10^{-6}}{r_\infty}(n_{1/2} + N) \text{ seconds.} \tag{4}$$

This model can be applied not only to predict the execution time of vectorized programs but also to develop optimal vector algorithms [65, 92]. Indeed, several algorithms (especially *divide and conquer* algorithms) consist of loops of different lengths, which are related to each other and can be treated as parameters of a vectorized program. Then (3) can be used for finding optimal values of these parameters that minimize the execution time of a program. For instance, in [92], the Hockney–Jesshope model of vector processing was applied to find a very fast vector algorithm for solving linear recurrence systems with constant coefficients.

5.3 Amdahl's law

One of the most important methods of analyzing the potential for parallelization of any algorithm is to observe how the algorithm can be divided into parts that can be executed in parallel and into those that have to be executed sequentially. More generally, different parts of an algorithm are executed with different speeds and use different resources of a computer to a different extent. It would be naive to predict the algorithm's performance by dividing the total number of operations by the average speed of the computer. Such a calculation would be at best a very crude estimate for single processor machines with a very simple memory hierarchy. To find a quality performance estimate, one should separate all parts of the algorithm that utilize the underlying computer hardware to a different extent. Significant initial work in this area was done by Amdahl [39, 40]. He established how slower parts of an algorithm influence its overall performance. Assuming that a given program consists of N floating point operations, out of which a fraction f is executed with a speed of V MFlops, while the remaining part of the algorithm is executed with a speed of S Mflops, and assuming further that the speed V is close to the peak performance while the speed S is substantially slower ($V \gg S$), then the total execution time can be then

expressed using the following formula:

$$t = f\frac{N}{V} + (1-f)\frac{N}{S} = N(\frac{f}{V} + \frac{1-f}{S}) \tag{5}$$

which can be used to establish the total execution speed of the algorithm as

$$r = \frac{N}{t} = \frac{1}{\frac{f}{V} + \frac{(1-f)}{S}} \text{ Mflops.} \tag{6}$$

From formula (5), it follows that

$$t > \frac{(1-f)N}{S}. \tag{7}$$

If the whole program is executed at the slower speed S, its execution time can be expressed as

$$t_s = \frac{N}{S}. \tag{8}$$

If the execution speed of the part f of the program can be increased to V then the performance gain can be represented as

$$\frac{t_s}{t_v} < \frac{N}{S} \cdot \frac{S}{N(1-f)} = \frac{1}{1-f} \tag{9}$$

This last formula is called Amdahl's Law and can be interpreted as follows: *The speed-up of an algorithm that results from increasing the speed of its fraction f is inversely proportional to the size of the fraction that has to be executed at the slower speed.*

In practice, Amdahl's Law *provides an estimate of the overall speed at which the algorithm can be executed.* Figure 4 illustrates the effect of the size of f, the fraction of calculation executed in vector speed, on the total performance of an algorithm to be executed on a vector computer, where $V = 1000$ MFlops (peak performance using a vector unit) and with $S = 50$ MFlops (performance of scalar calculations).

It can be observed that a relatively large $f = 0.8$ results in an average speed of *o*nly 200 MFlops. In computational practice, this result illustrates an intuitive fact: to reach the highest possible level of performance of a given program, the most important parts are those that are the slowest. Moreover, even a relatively small fraction of a slow code can substantially reduce the overall speed achieved by the whole program – see the rapid decrease of the performance graph from $f = 1.0$ to $f = 0.8$. A typical example of how the speed of the slowest parts of the program influences the overall

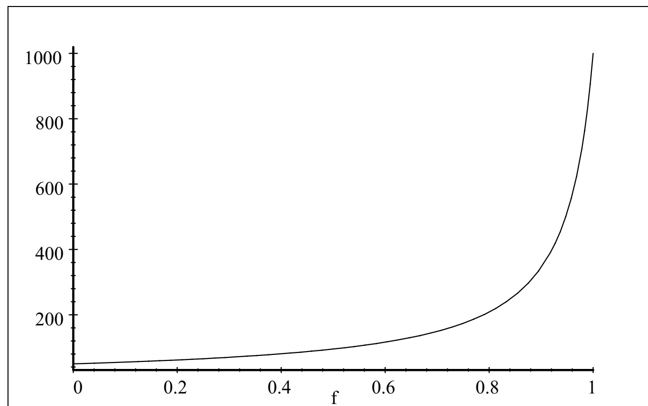

Figure 4: Amdahl's Law (performance in MFlops) for $V = 1000$ and $S = 50$ (increasing fraction f)

performance of the vector computer would be a notorious case of recurrent formulas [84, 91].

Formula (6) can be easily generalized to the case of a program consisting of n separate parts $(A_1, \ldots, A_n)$ that are executed at different speeds. Here, N_j operations of part A_j are executed with speed r_j MFlops, and $N = \sum_{j=1}^{n} N_j$. Thus the average speed of the whole program can be expressed as:

$$r = \frac{N}{\sum_{j=1}^{n} \frac{N_j}{r_j}} \quad \text{Mflops}, \tag{10}$$

and again, the average performance will be determined primarily by the speed of its slowest parts.

For a parallel program, the total execution time, understood as the sum of execution times on all processors, is usually larger than the total time used by the same algorithm executed on a single processor (not considering special cases, where the effects of job splitting among multiple processors with large memories affect the total performance by freeing the program from the influence of memory related bottlenecks, which occur on a single processor machine). However, as specified above, the main goal of parallel computing is to reduce the humanly observable (wall-clock) execution time of the algorithm. The increase of the total time is the price to be paid for

the reduction of the wall-clock time. If an algorithm can be divided into p equal parts that can be executed concurrently on p processors then it is conceivable that the parallel execution time will be $1/p$ of the single processor time. Such a situation is practically impossible for all non-trivial algorithms. Since the sequential parts of an algorithm are the "slowest," Amdahl's law dictates that these parts have the most serious negative impact on the overall performance.

More precisely, in the case of a parallel machine with p homogeneous processors, the speedup s_p over the sequential algorithm achieved due to p processors can be expressed as:

$$s_p = \frac{t_1}{t_p}. \tag{11}$$

where t_j denotes time of execution of the algorithm on j processors. Assuming that fraction f of the algorithm can be divided into p parts and ideally parallelized (executed at exactly the same time t_1/p on p processors), the remaining $1 - f$ of operations cannot be parallelized and thus have to be executed on a single processor. The total execution time of this algorithm on p processors can be expressed as:

$$t_p = f\frac{t_1}{p} + (1-f)t_1 = \frac{t_1(f + (1-f)p)}{p}.$$

Therefore the speedup s_p is equal to

$$s_p = \frac{p}{f + (1-f)p}. \tag{12}$$

Obviously, $f < 1$, and therefore the following inequality is true

$$s_p < \frac{1}{1-f}. \tag{13}$$

This inequality (13) is known as *Amdahl's Law for parallel computing.* It states that the *speedup achievable through parallel computing is bound by the value that is inversely proportional to the fraction of the code that has to be executed sequentially.*

Similarly to the general Amdahl's Law [39] the above considerations provide a simple way to initially assess the expected parallel performance of an algorithm. If $f = 0.9$, so that 90% of an algorithm can be ideally parallelized, and if $p = 10$, the formula (12) gives the result that the speedup cannot exceed 5.

While the situation presented here is obviously highly idealized, it allows to conceptualize the effects of load imbalance. In the case of a program executing on two processors, where at a certain moment processor 1 has completed its work while processor 2 is still executing its part of the parallel code, the still-to-be-executed part of the code on processor 2 becomes a serial part of the parallel program and formula (12) applies. In general, in the case of a parallel program executing on p processors, due to the load imbalance at various times t_j, different numbers p_j of processors are working on the code while the remaining $p - p_j$ processors are idle, therefore:

$$S_p = \frac{1}{\sum_{i=1}^{k-1} \frac{f_i}{p_i} + f_k} \quad \text{where} \quad \sum_{i=1}^{k} f_i = 1. \tag{14}$$

This case can be treated as the generalized case of Amdahl's Law, where various fractions of the code can be executed by a different number of processors. Obviously, each situation, when only a number of processors smaller than p is used, leads to the degradation of overall performance, which indicates how important load balancing is for the parallel program performance.

An obvious idealization lies in Amdahl's law because it only takes into account a fixed problem size. More often, the problem size is expected to scale with the number of processors (one of the important reasons for applying a parallel computer with distributed memory is the fact that with every processor additional memory is added to the system thus allowing solution to a larger problem). Thus, in [57], Gustafson proposed an alternative to Amdahl's Law. *Rather than asking how fast a given serial program would run on a parallel machine, he asks how long a given parallel program would have taken to run on a serial processor* [58]. Let t_s and t_p denote the serial and the parallel time spent on a P processor parallel machine, respectively, and let $t_s + t_p = 1$, then *the scaled speedup* of the parallel program will be given by

$$s = \frac{t_s + P t_p}{t_s + t_p} = t_s + P t_p = P + (1 - P) t_s.$$

This model is somewhat more realistic in its predictive power. Its usability was presented in [58].

Theoretical considerations presented here as well as in the literature show that, in computational practice, a very large number of factors will influence the parallel execution time of an algorithm. Most of these factors are much more likely to degrade the performance gains of parallelization rather than to augment them. Some of the factors that should be taken into consideration are listed below:

- The algorithm itself must be parallelizable and the data set to which it is to be applied must be such that an appropriately large number of processors can be applied.
- Overheads related to synchronization and memory access conflicts will lead to performance deterioration.
- Load balancing is usually rather difficult to achieve and the lack of it results in performance deterioration.
- Creation of algorithms that can be used on multiple processors often leads to the increase of computational complexity of the parallel algorithm over the sequential one.
- Dividing the data among multiple memory units may reduce the memory contention and improve data locality resulting in performance improvement.

In summary, there are a number of theoretical models that can be applied to predict the performance of parallel algorithms. Unfortunately, all of them are highly idealized and thus are limited to supplying general performance expectations as well as to pointing out the most important issues that should be taken into account when a parallelization of a sequential algorithm and an existing sequential code is attempted. When a more detailed performance prediction is required, then a particular model for a given problem and its characteristics as well as the hardware, on which it is to be executed, have to be taken into account.

6 Programming parallel computers

Moving from theoretical considerations to the computational practice, one must consider a number of issues involved in programming parallel computers. This introduction to them will follow the above described levels of parallelization and start from the parallelization inside of a single processor as well as a brief note about language based parallelization. It will be followed by the discussion of writing codes for shared memory and distributed memory computers, parallelization of existing codes and library-based parallelization.

6.1 Optimizing compilers

Optimizing compilers [100, 101] offer several optimization levels. They transform a code according to the specified option(s). These transformations are cumulative: each higher level retains the transformations / optimizations of the previous level. The available optimization levels are typically as follows (while the levels described below are not specific to any particular hardware architecture and to any computer vendor, similar levels can be found across most of them):

1. *Machine-dependent scalar optimization, usually a default*, which fully exploits the machine's scalar functional units and registers.

2. *Basic block machine-independent scalar optimization* works at the local basic-block level. A basic block is a branch-less sequence of statements ending with a conditional or unconditional branch. At this level a compiler uses such techniques as assignment substitution, elimination of common subexpressions, constant propagation and folding.

3. *Program block machine-independent scalar optimization* works at the global program-unit level (a subroutine, function or main section) using eliminations of redundant assignments, dead-codes and hoisting as well as sinking scalar and array references.

4. *Vector optimization* (if applicable) improves the performance of programs that manipulate arrays. Consider a loop adding the corresponding elements of two arrays. Within this level of optimization, the vector CPU (if available) can add groups of array elements utilizing a single machine instruction. For example, the following simple loop

   ```
   do j=1,100
   z(j) = x(j) + y(j)
   end do
   ```

   will be transformed to a vector instruction

   ```
   z(1:100)=x(1:100)+y(1:100)
   ```

   When the loop length is greater than the vector length or when it is unknown, the compiler will generate a loop that repeats the above vector instruction.

5. *Parallel optimization* allows to spread work across multiple CPUs and typically analyzes the loop structure of the code. In case of parallel machines with vector processors, the inner loops are vectorized while the outer loops are parallelized.

All optimization levels but the last one should generate code optimal for a given processor and the memory structure of a given machine. The last level of optimization is typically available on shared memory parallel computers (most vendors of parallel computers provide Fortran and C compilers capable of code parallelization). Early successful work on optimizing compilers for high performance (parallel) computers can be traced to the Cray optimizing compilers for Fortran, and Kuck and Associates parallelizing compilers for Fortran and C [7]. While this statement is open to discussion, there is little question that Cray's Fortran optimizing compiler was one of the best (in terms of the quality of generated code) products existing at the times of Cray Y-MP and Cray C-90 supercomputers. Unfortunately, its success can be traced to the relative simplicity of the Cray architecture, few (up to 64, but typically 8 or 16) vector processor and a large global memory. As soon as the modern workstations with hierarchical memory are considered, the performance of optimizing compilers becomes less satisfactory. Experiments show a substantial performance degradation of a simple matrix multiplication operation on an MIPS and Alpha processor based computers from SGI and DEC, when the matrix size increased past $n = 1000$ [19, 18]. These results indicate that optimizing codes for modern architectures with multiple levels of data latency and multiple sizes of intermediate memory layers is a complicated endeavor. Projects like ATLAS [99] attempt to remedy these problems for linear algebraic operations (see also 6.7.1) but, in general, a lot more work is required. The situation is even worse for code parallelization. While parallelizing compilers can deal with microparallelization and matching the code with multiple execution pipelines of modern processors, as well as with parallelization of simple loops, they have problems with parallelization of complicated structures of the program. The situation becomes particularly complicated when parallelization requires a serious restructurization of the program. It is arguable that while the compiler based parallelization will play an increasingly important role in the implementation of algorithms (it is, for instance, claimed that with the increasing power of computers an even larger "window" of the code can be considered at once thus increasing compilers' ability to analyze and optimize the machine code), it will be mostly responsible for low level parallelization and, at least for some time to come, cannot be relied on as a method for building parallel programs. In other

words, implementers have to do the work themselves and the parallelizing / optimizing compiler can fine-tune the results of their work.

6.2 Language based parallelization

The second approach to parallelization is based on the language constructs. Two scenarios can be distinguished. First, constructs inside of the language support various possible levels of parallelism, e.g. various versions of Fortran (primarily High Performance Fortran) and SISAL. Second, language constructs are geared toward high level of parallelism, e.g. Ada and Java. Since Fortran was one of the early languages applied in large scale scientific computing, efforts were undertaken to extend the Fortran 77 definition to add more constructs supporting high performance and parallel computing. These additions varied from the vector and matrix oriented operations in Fortran 90/95 standards [11, 73] to a more thorough support of parallelism in the High Performance Fortran [68]. Interestingly, thus far neither of these approaches gained widespread popularity among parallel program developers (for more details about High Performance Fortran, see chapter **??**). SISAL is an example of an attempt at bringing functional programming, which is said to be one of the better approaches to parallel program design, to scientific computing [12, 56]. The most interesting feature of SISAL is the fact that it combines the imperative and functional programming paradigms. Although initial results were quite encouraging [26, 22], SISAL has practically disappeared after a few years of development.

Ada was originally designed to support concurrency and thus included support for most functions necessary to develop and implement parallel programs. However, for a variety of reasons summarized in [85], it has never been seriously considered for scientific parallel computing application development. For instance, although until recently the US Department of Defense required all of its computing to be done in Ada, numerically intensive codes, e.g. ocean modeling applications, were developed and implemented in Fortran and translated into Ada by a separate group of programmers when ready to be turned into a production environment. Finally, there is Java, which also can be used to naturally develop parallel programs (e.g. through the application of multi-threading). Since it is a relatively new language, it is still unclear how much popularity it will gain in the scientific computing community. The main disadvantage of Java seems to be its widely perceived lack of efficiency (which should not be viewed as a problem with the language itself since it was not designed for that purpose). Java Grande "community" (among others) attempts to remedy this problem [27, 49] and time will tell

how successful their efforts will be (for more detailed treatment of Java as a language for parallel computing, see chapter ??).

6.3 Shared memory parallel algorithms and OpenMP

Parallel architectures based on a shared memory have now become commonplace and usually offer more than just a few processors. Until quite recently, each vendor has provided its own set of commands to support writing parallel programs. All these approaches were quite similar and consisted of directives for managing parallel code execution; e.g. loop parallelizing directives, locks, barriers and other synchronization primitives etc., inserted into codes written in Fortran or C. It was only recently that OpenMP [30] emerged as a standard for code parallelization for shared memory parallel computers. While OpenMP provides support for three basic aspects of parallel computing:

- specification of parallel execution,
- communicating between multiple thread,
- expressing synchronization between threads,

and could be potentially used to support parallelism on any computer architecture, it is best suited for shared memory environments. OpenMP directives satisfy the following format:

```
!$omp directive name   optional clauses
```

Such approach allows to write the same code for both single-processor and multiprocessor platforms. Simply, compilers which do not support OpenMP directives or those that are working in a single-processor mode treat them as comments.

The OpenMP uses the fork-join model of parallel execution. A program starts execution as a single process, called the *master thread* of execution, and executes sequentially until the first parallel construct is encountered. Then the master thread spawns a specified number of "slave" threads and becomes a "master" of the team. All statements enclosed by the parallel construct are executed in parallel by each member of the team. Several directives accept clauses that allow a user to control the scope attributes of variables for the duration of the construct (e.g. shared, private, reduction, etc.). The following code can serve as an example of a simple parallel program [10]:

```
     print *,'#procs='
     read *,p
     call omp_set_num_threads(p)
!$omp parallel shared(x,npoints) private(iam,np,ipoints)
     iam=omp_get_thread_num()
     np=omp_get_num_threads()
     ipoints=npoints/np
     call work_on_subdomain(x,iam,ipoints)
!$omp end parallel
```

Each thread in the parallel region determines what part of the global array `x` to work on. The code contains calls to OpenMP routines: `omp_set_num_threads` which sets the number of threads in a parallel construct, `omp_get_thread_num` which returns the number of a calling thread and `omp_get_num_threads` which returns the number of threads in a parallel region. In this code, each processor will execute the `work_on_subdomain` routine.

In the case of shared memory machines, the most common work-sharing construct within a parallel region is the *do-construct*, which distributes iterations of the do-loop among available working threads.

```
!$omp parallel do
      do j=1,n
        ........
      end do
!$omp end parallel do
```

The OpenMP compiler *do-construct* divides the iterations of a do-loop into subranges of the j index and hands them to different threads, which execute them in parallel.

Sometimes, parallelism can be expressed by means of the *sections* construct. In the example below, functions `proc_one()` and `proc_two()` are executed in parallel, which allows high level parallelization based on heterogeneous tasks and enables the OpenMP written codes to be applied also beyond the simple loop parallelization.

```
!$omp parallel sections
!$omp section
      call proc_one()
!$omp section
      call proc_two()
!$omp end parallel sections
```

Finally, numerical integration can serve as a slightly more complicated example of utilizing OpenMP for parallel computing:

$$\int_a^b f(x)dx \approx h\left(\frac{f(x_0)}{2} + f(x_1) + \ldots + f(x_{n-1}) + \frac{f(x_n)}{2}\right) \qquad (15)$$

where $h = \frac{b-a}{n}$ and $x_i = a + ih$, $i = 0, \ldots, n$. An OpenMP code to complete this task would look like this:

```
      h=(b-a)/n
!$omp parallel do private(x) reduction(+:sum)
      do j=1,n-1
        x=a+j*h
        sum=sum+f(x)
      end do
!$omp end parallel do
      sum=h*(sum+0.5*(f(a)+f(b)))
```

In this example, the `reduction(+:sum)` clause is used. It instructs the compiler that the variable `sum` is the target of a sum reduction operation.

The examples presented above as well as other examples are included to give the reader a general feeling for what a code written for a given environment looks like. For further details on OpenMP, consult [8, 10, 30]. Similarly, the references should be consulted for the details concerning the remaining tools and environments.

6.4 Distributed memory parallelization

As indicated above (sections 3.5.2 and 4.3), parallelization for distributed memory computers typically consists of dividing the program into separate computational units that work independently and communicate by exchanging messages. To illustrate the basics of such an approach, consider a simple example of distributed computation of the vector norm

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

Assume that there exist integers p and q such that $pq = n$ and that the code is to be executed on a parallel computer. One of possible approaches to parallelizing the problem is based on the master-slave approach. Here, the program consists of two types of tasks: *master task* and *slave tasks* T_i, $i = 1, \ldots, p$. The following pseudo-code can illustrate this approach:

MASTER

1. Get n, x and choose p and q.
2. Spawn p slave tasks T_i, $i = 1, \ldots, p$.
3. Send q to all slave tasks T_i.
4. For $i = 1, \ldots, p$: send numbers $x_{(i-1)q+1}, \ldots, x_{iq}$ to T_i.
5. Set $sum \leftarrow 0$; for $i = 1, \ldots, p$: receive "partial sum" from a slave task and assign it to a variable s and $sum \leftarrow sum + s$.
6. Assign $sum \leftarrow \sqrt{sum}$

SLAVE

1. Receive from "MASTER" the value of q.
2. Receive from "MASTER" q numbers $y_1, \ldots, y_q$.
3. Calculate
$$s \leftarrow \sum_{i=1}^{q} y_i^2.$$
4. Send s to "MASTER".

While the master-slave based computing is only one of the possible approaches to distributed memory parallel computing, it illustrates the general idea of dividing work into independent tasks and coordinating computations through message passing. The two most popular environments supporting this mode of computing will now be considered.

6.4.1 PVM and MPI

Parallel Virtual Machine (PVM) has been developed in 1991 by a group of researchers at the University of Tennessee. It is a distributed-memory tool [41] designed to develop parallel applications on networks of heterogeneous computers. It allows to use such a heterogeneous environment as a single computational resource. PVM consists of a daemon software that should be run on each node of the parallel virtual machine, a console and a library, which provides subroutines for process creation and message passing. Currently this library supports APIs for Fortran and C/C++. It should be noted that in its philosophy the PVM environment can be considered an “interactive” one. In its typical mode of operation, the PVM user works from

the console, initializes the heterogeneous environment, instantiates the PVM daemons on all machines that participate in the virtual computer and starts the execution of the main program. Experiments show that PVM sometimes do not work well in batch processing environments [83, 82, 81].

What follows is an example of the PVM code that calculates the integral (15) using the master-slave approach.

```
main() {
 int myid, nprocs, howmany;
 int tids[10];
 float a,b,h,lsum,tol;
 tids[0]=pvm_mytid();
 myid=pvm_joingroup("integral"); /* join the group */
 if (myid==0) /* i'am the first task - the master task*/
   { a=0.0;
     b=3.141569;
     printf("How many processors ? ");
     scanf("%d",&nprocs);
     /* spawn nprocs-1 tasks */
     howmany=pvm_spawn("int01",(char**)0,0,"*",nprocs-1,&tids[1]);
     /* wait for other processes */
     while(pvm_gsize("integral")!=nprocs)
       ;
     /* send data */
     pvm_initsend(PvmDataDefault);
     pvm_pkint(&nprocs,1,1);
     pvm_pkfloat(&a,1,1);
     pvm_pkfloat(&b,1,1);
     pvm_bcast("integral",10);
   }
 else /* I'm a slave */
   { pvm_recv(-1,10);
     pvm_upkint(&nprocs,1,1);
     pvm_upkfloat(&a,1,1);
     pvm_upkfloat(&b,1,1);
   }
 /* find the approximation */
  h=(b-a)/nprocs;
  a=a+h*myid;
  b=a+h;
```

```
  lsum=0.5*h*(f(a)+f(b));
 /* gather partial results */
  pvm_reduce(PvmSum,&lsum,1,PVM_FLOAT,20,"integral",0);
  if (myid==0)
    { printf("%f\n",lsum);
    }
  pvm_barrier("integral",nprocs);
  pvm_exit();
}
```

It should be noted that in this as well as in the previous example of calculating the norm of the vector, the master is performing tasks that are completely different from the tasks of the slave processes. This does not need to be the case. If the computational workload of each of the slave processes is substantially larger than that of the master, the master should also perform some work (possibly a smaller overall amount than the slaves), while waiting for the slaves to complete their tasks. This approach, while slightly more difficult to implement, helps to improve the load balancing (the master is not idle) and thus results in a better overall performance (see section 5.3).

Message Passing Interface (MPI) has been developed in 1993 [77] by researchers from Argonne National Laboratory. Over time, it has become a de-facto standard for message passing parallel computing (superseding PVM, which is slowly becoming extinct). MPI provides an extensive set of communication subroutines including point-to-point communication, broadcasting and collective communication. It has been implemented on a variety of parallel computers including massively parallel computers, clusters and networks of workstations. Due to its popularity, a number of open source and commercial tools and environments have been developed to support MPI based parallel computing [55].

As an example, consider again the integration problem. However, this time, it will be solved using the SPMD model. Here each processor calculates its own part of the integral and then all of them exchange partial sums. At the end of the process, each processor calculates and contains its own local copy of the integral. While it may seem unreasonable to assume that each processor needs to calculate its own copy of the integral, one can assume that this is just a part of a larger code and these integral values are used by each processor independently in subsequent calculation.

```
    integer ierr, myid, numprocs, rc, j
    real*8 x, sum, global_sum, a, b, my_a, my_b, my_h, h
```

```
*
      read *,a,b
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      h=(b-a)/numprocs
      my_a=a+myid*h
      my_b=my_a+h
      sum=h*(sum+0.5*(f(my_a)+f(my_b)))
* collect all the partial sums
      call MPI_REDUCE(sum,global_sum,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     $ MPI_COMM_WORLD,ierr)
* node 0 prints the answer.
      if (myid .eq. 0) then
        print *,'Result is ',global_sum
      endif
      call MPI_FINALIZE(rc)
      stop
      end
```

6.5 Shared-distributed memory environments

As indicated above (Section 3.5), there are a number of approaches to implementing parallel algorithms on shared-distributed memory computers. Such a parallelization can be done by treating such a computer as a virtual shared memory environment (and use, for instance, OpenMP), as a distributed memory machine (and use MPI). However, to be able to use the machine to the fullest extent, one may want to consider a mixed approach. A modification of the above code for the calculation of the integral that utilizes jointly MPI and OpenMP can illustrate such an approach.

```
      parameter (n=1000)
      integer ierr, myid, numprocs, rc, j
      real*8 x, sum, global_sum, a, b, my_a, my_b, my_h, h
*
      read *,a,b
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      my_h=(b-a)/numprocs
```

```
    my_a=a+myid*my_h
    my_b=my_a+my_h
    h=(my_b-my_a)/n
!$omp parallel do private(x) reduction(+:sum)
    do j=1,n-1
      x=my_a+j*h
      sum=sum+f(x)
    end do
!$omp end parallel do
    sum=h*(sum+0.5*(f(my_a)+f(my_b)))
* collect all the partial sums
    call MPI_REDUCE(sum,global_sum,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
   $ MPI_COMM_WORLD,ierr)
* node 0 prints the answer.
    if (myid .eq. 0) then
      print *,'Result is ',global_sum
    endif
    call MPI_FINALIZE(rc)
    stop
    end
```

Here, the division of the workload is initially done in the MPI environment. Then, on each shared memory computer, the OpenMP directives are used to perform loop parallelization. Obviously, such an approach is slightly more complicated and more tedious to implement but, as a result, the code matches the underlying hardware architecture, which may lead to further performance gains.

6.6 Parallelization of existing codes

Consider the situation when an existing sequential code is to be parallelized. As stated above, at the present time one cannot count on the parallelizing compiler (or any other tools) to mechanize and substantially simplify the process of restructuring an old code for parallel processing. Most of the necessary work must be done manually and must be based on a clear understanding of the inner-workings of the code. While each case has to be treated separately and will involve its own particular challenges, there are some general rules that follow from what has been said above. First, the selection of the approach: while an OpenMP loop parallelization may be the best solution (and may be the easiest to implement) in the case of the

code that is to run on a shared memory parallel computer, this approach may not be the best in the long run. It seems clear that in the near future most of the really large computers are going to be based on the distributed memory paradigm and may, or may not, consist of shared memory nodes. If so then the rational advice is to seriously consider an MPI-based parallelization. The MPI-based approach may be combined with the OpenMP when a shared-distributed memory hardware is to be used, but this may be an overkill not worth the effort. While there exist a number of other tools and environments that can be utilized, and each one of them has its own merits, it is arguable that **if** the goal of the project is to develop and apply software then researchers should stay with proven technology, which represents the state of the art at the time. For this and other reasons presented above, at this time Java does not seem to be ready to appear in the solvers for computationally intensive tasks.

Second, an important issue is the analysis of the existing code. Here the lessons learned from the application of Amdahl's Law play an important role. It follows from it that attention must be paid to the most computationally intensive and therefore most time consuming parts of the code. It makes no difference whether a part of the code, which takes only 10% of the total time, is perfectly parallelized, if the remaining parts of the code are not. Therefore, the early stages of the code analysis should consist of benchmarking and time-profiling. As soon as a well-developed profile is created, it becomes clear where to focus further efforts.

Finally, one more area needs to be seriously considered. While an old code may be used for many years, there exists a possibility that this code, or its parts, can be replaced by the existing modules stored, among others, in the Netlib repository [6], the ACM TOMS [3] library or announced, among others, on the NA Digest forum. Recent years have witnessed a rapid development of codes for the efficient parallel solution of a wide variety of mathematical problems. These codes have been implemented and tested on a number of parallel computers and, very often, they are very high quality both from the point of view of numerical properties as well as parallel performance. This approach can be called *library-based parallelization*. (In the next section, a discussion of software available from one of the more successful projects in the area of numerical linear algebra for dense matrices is presented.) It needs to be stressed, that the above discussion contains only a fragment of what is available; see Chapters **??**, **??**, **??** for more examples of existing software. Overall, before an attempt is made to write a parallel code to solve a given problem, a thorough search should be conducted for existing software because chances are that a ready-to-use routines are already

available.

6.7 Library-based parallelization

There are a great many applications for which software has been implemented and which can be used when solving a particular computational problem. This can be done on two levels. First, there exist complete software packages that can be utilized to solve problems in parallel, e.g. parallel PDE solver ELLPACK [66, 89]. There exist also several packages designed for supporting parallel computing involving sparse matrices [47, 54, 102]. While some of these environments are definitely state-of-the-art and using them is preferable to developing the code oneself, there are problems that may not fit well enough into the existing software. There are also new algorithms that have to be implemented. In this case, there may exist libraries of "building blocks" that should be used in the process. What follows is a brief introduction to one of the more robust libraries supporting the development of high performance and parallel codes involving matrix operations.

6.7.1 BLAS and LAPACK

In the area of linear algebraic computations for dense and band structured matrices, there exists a de facto standard for writing high performance software. More precisely, there exists a collection of interdependent software libraries that became the standard tool for dense and banded linear software implementation (even though the latter has been recently challenged by the work of F. Gustafson [59]).

The first step in the general direction took place in 1979, when the BLAS (Basic Linear Algebra Subroutines) standard was proposed [71]. Researchers realized that linear algebra software (primarily for dense matrices) consists of a number of basic operations (e.g. vector scaling, vector addition, dot product, etc.). These fundamental operations have been defined as a collection of Fortran 77 subroutines. The next two steps took place in 1988 and 1990, respectively, when the collection of matrix-vector and matrix-matrix operations have been defined [38, 37]. These two developments can be traced to the hardware changes happening at this time. The introduction of hierarchical memory structures resulted in the increasing need for the development of algorithms that would support data locality (move the *block* of data once, perform all the necessary operations on it and move a data back to the main memory and proceed with the next data block). It was established that to achieve this goal one should rewrite linear algebra codes in terms of

block operations and such operations can be naturally represented in terms of matrix-vector and matrix-matrix operations.

The BLAS routines were used in the development of linear algebra libraries that solved a number of the standard problems. Level 1 BLAS (vector oriented operations) was used in the development of the LINPACK [36] and EISPACK [53] libraries devoted to the solution of linear systems and eigenproblems. The main advantages of these libraries were: the clarity and readability of the code, its portability as well as the possibility of hardware oriented optimization of the BLAS kernels.

The next step was the development of the LAPACK library [15], which "combined" the functionalities available in the LINPACK and EISPACK libraries. LAPACK was based on utilizing level 3 BLAS kernels, while the BLAS 2 and 1 routines were used only when necessary. It was primarily oriented toward single processor high performance computers with vector processors (e.g. Cray, Convex) or with hierarchical memory (e.g. SGI Origin, HP Exemplar, DEC Alpha workstations etc.). The LAPACK was also designed to work well with shared memory parallel computers, providing parallelization inside the level 3 BLAS routines [40]. Unfortunately, while this performance was very good for the solution of linear systems (this was also the data used at many conferences to illustrate the success of the approach), the performance of eigenproblem solvers (for both single processor and parallel machines) was highly dependent on the quality of the underlying BLAS implementation and very unsatisfactory in many cases [19, 18].

In algebraic notation, the BLAS operations have the following form (detailed description of the BLAS routines can be found in [15, 39]):
Level 1: vector-vector operations:

- $y \leftarrow \alpha x + y$, $x \leftarrow \alpha x$, $y \leftarrow x$, $y \leftrightarrow x$, $dot \leftarrow x^T y$, $nrm2 \leftarrow \|x\|_2$, $asum \leftarrow \|re(x)\|_1 + \|im(x)\|_1$.

Level 2: matrix-vector operations:

- matrix-vector products: $y \leftarrow \alpha Ax + \beta y$, $y \leftarrow \alpha A^T x + \beta y$
- rank-1 update of a general matrix: $A \leftarrow \alpha xy^T + A$
- rank-1 and rank-2 update of a symmetric matrix: $A \leftarrow \alpha xx^T + A$, $A \leftarrow \alpha xy^T + \alpha yx^T + A$,
- multiplication by a triangular matrix: $x \leftarrow Tx$, $x \leftarrow T^T x$,
- solving a triangular system of equations: $x \leftarrow T^{-1}x$, $x \leftarrow T^{-T}x$.

Level 3: matrix-matrix operations:

- matrix-matrix products: $C \leftarrow \alpha AB + \beta C$, $C \leftarrow \alpha A^T B + \beta C$, $C \leftarrow \alpha AB^T + \beta C$, $C \leftarrow \alpha A^T B^T + \beta C$
- rank-k and rank-$2k$ update of a symmetric matrix: $C \leftarrow \alpha AA^T + \beta C$, $C \leftarrow \alpha A^T A + \beta C$, $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$,
- multiplication by a triangular matrix: $B \leftarrow \alpha TB$, $B \leftarrow \alpha T^T B$, $B \leftarrow \alpha BT$, $B \leftarrow \alpha BT^T$,
- solving a triangular system of equations: $B \leftarrow \alpha T^{-1} B$, $B \leftarrow \alpha T^{-T} B$, $B \leftarrow \alpha BT^{-1}$, $B \leftarrow \alpha BT^{-T}$.

Observe that each operation from the BLAS 2 library can be expressed in terms of BLAS 1 operations. Consider, for instance, a matrix-vector multiplication

$$y \leftarrow \alpha Ax + \beta y \tag{16}$$

can be conceptualized in terms of a sequence of dot-products (routine _DOT), vector scalings (_SCAL) or vector updates (_AXPY).

$$\begin{array}{ll} z_k \leftarrow A_{k*}x, \text{ for } k = 1, ..., m & (\text{_DOT}) \\ y \leftarrow \beta y & (\text{_SCAL}) \\ y \leftarrow y + \alpha z & (\text{_AXPY}) \end{array} \tag{17}$$

Observe also that regardless of the fact that the above described algorithm and procedure (16) are equivalent, the application of the BLAS 2 based approach can substantially reduce the amount of processor-memory communication and thus reduce the overall execution time. Similarly, operations represented by the BLAS 3 routines can be expressed in terms of lower level BLAS. For instance, operation

$$C \leftarrow \alpha AB + \beta C \tag{18}$$

can be expressed as

$$C_{*k} \leftarrow \alpha AB_{*k} + \beta C_{*k}, \quad \text{for } k = 1, ..., n, \tag{19}$$

which is a sequence of operations denoted by (16). It should be also noted that, as in the case of replacing level 1 BLAS operations by level 2 BLAS, the application of (18) instead of (19) reduces the total amount of processor-memory communication. More precisely, to illustrate the advantages of the

BLAS	loads and stores	flops	ratio
$y \leftarrow y + \alpha x$	$3n$	$2n$	$3:2$
$y \leftarrow \alpha Ax + \beta y$	$mn + n + 2m$	$2m + 2mn$	$1:2$
$C \leftarrow \alpha AB + \beta C$	$2mn + mk + kn$	$2mkn + 2mn$	$2:n$

Table 1: BLAS: memory references and arithmetic operations

	MFlops	sec.
BLAS 1	93.81	21.32
BLAS 2	355.87	5.62
BLAS 3	1418.44	1.41

Table 2: Matrix multiplication on the Pentium III 866MHz

application of higher level BLAS, consider the total number of arithmetical operations and the amount of data exchanged between the processor and memory. Table 1 [39] depicts the ratio of the number of processor-memory communications to the number of arithmetical operations for $m = n = k$.

The higher the level of BLAS, the more favorable the ratio becomes. (The number of operations performed on data increases relative to the total amount of data movement.) This has a particularly positive effect in the case of hierarchical memory computers (see also Section 3.2).

To illustrate that this course of action plays an important role not only for "supercomputers" but also for more "ordinary" architectures, Table 2 presents processing speed (in MFlops) achieved during the completion of the task $C \leftarrow \alpha AB + \beta C$ using BLAS 1, 2 and 3 kernels (utilizing algorithms 17 and 19) for $m = n = k = 1000$ on a single-processor PC with Intel Pentium III 866MHz processor with 512MB RAM.

There are two ways of using BLAS routines in parallel computing. First, very often, BLAS routines are parallelized by the computer hardware vendors. For instance, a call to the level 3 BLAS routine _GEMM may result in parallel execution of matrix-matrix multiplication. Any code that utilizes _GEMM will automatically perform this operation in parallel. While some computer vendors spend considerable amount of time and resources to deliver highly optimized BLAS kernels (routines accumulated in Cray's *scilib* library were one of the best in delivered performance, while currently IBM's *ESSL* library is also very well optimized), this does not have to be the case. In addition, only some of BLAS kernels are parallelized (one of the typical

and very important exceptions are routines for symmetric matrices stored in a compact form) and they are typically parallelized for shared memory environments only (for an example of problems encountered in parallelization of BLAS kernels, see [17]). In short, parallel performance of BLAS routines cannot be taken for granted (especially since they are primarily optimized for single processor performance in the hierarchical memory environment). Taking this into account BLAS kernels should be rather utilized to develop parallel programs where the BLAS routines will run on separate processors. To illustrate the main idea behind such an approach, consider matrix update procedure based on the formula $C \leftarrow \alpha AB + \beta C$, which can be rewritten as:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \alpha \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} + \beta \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \tag{20}$$

Applying the definition of matrix multiplication, the following block algorithm to calculate matrix C is obtained.

$$\begin{array}{ll} C_{11} \leftarrow \alpha A_{11} B_{11} + \beta C_{11} & /1/ \\ C_{11} \leftarrow \alpha A_{12} B_{21} + C_{11} & /2/ \\ C_{12} \leftarrow \alpha A_{11} B_{12} + \beta C_{12} & /3/ \\ C_{12} \leftarrow \alpha A_{12} B_{22} + C_{11} & /4/ \\ C_{21} \leftarrow \alpha A_{22} B_{21} + \beta C_{21} & /5/ \\ C_{21} \leftarrow \alpha A_{21} B_{11} + C_{21} & /6/ \\ C_{22} \leftarrow \alpha A_{22} B_{22} + \beta C_{22} & /7/ \\ C_{22} \leftarrow \alpha A_{21} B_{12} + C_{22} & /8/ \end{array} \tag{21}$$

Observe that this algorithm allows for parallel execution of operations /1/, /3/, /5/, /7/ and in the next phase of operations /2/, /4/, /6/, /8/. Obviously, it is desirable to divide large matrices into a larger number of blocks (e.g. to match the number of available processors). The order of operations may also need to be adjusted to reduce the memory access conflicts. The application of this approach can be illustrated by the block-Cholesky method for solving systems of linear equations for symmetric positive definite matrices.

It is well known that there exists a unique decomposition for such matrices

$$A = LL^T, \tag{22}$$

where L is a lower triangular matrix. There exists also a simple algorithm for determining the matrix L with an arithmetical complexity of $O(n^3)$. Its analysis allows one to see immediately that it can be expressed in terms

of calls to the level 1 BLAS. Consider how it can be translated into block operations expressed in terms of level 3 BLAS. Formula (22) can be rewritten in the following way:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{pmatrix} \quad (23)$$

Thus

$$A = \begin{pmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T & L_{11}L_{31}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T & L_{21}L_{31}^T + L_{22}L_{32}^T \\ L_{31}L_{11}^T & L_{31}L_{21}^T + L_{32}L_{22}^T & L_{31}L_{31}^T + L_{32}L_{32}^T + L_{33}L_{33}^T \end{pmatrix}$$

After the decomposition $A_{11} = L_{11}L_{11}^T$, which is the same decomposition as the original one but of smaller size, appropriate BLAS 3 kernels can be applied in parallel to calculate matrices L_{21} and L_{31} by applying in parallel equalities $A_{21} = L_{21}L_{11}^T$ and $A_{31} = L_{31}L_{11}^T$. In the next step, equation

$$A_{22} = L_{21}L_{21}^T + L_{22}L_{22}^T,$$

can be used. Thus the decomposition $L_{22}L_{22}^T$ for the matrix $A_{22} - L_{21}L_{21}^T$ is used to calculate the matrix L_{22}. Finally, L_{32} can be found from

$$L_{32} = (A_{32} - L_{31}L_{21}^T)(L_{22}^T)^{-1}.$$

In a similar way, subsequent block columns of the Decomposition can be calculated. Finally, it should be noted that the parallelization of the matrix multiplication presented here as well as the Cholesky decomposition are examples of the divide-and-conquer method, which is one of the popular approaches to algorithm parallelization.

6.7.2 BLACS, PBLAS and ScaLAPACK

At the time when the LAPACK project was completed, it became clear that there is a need to develop similar software to solve linear algebraic problems on distributed memory architectures. Obviously, this could have been done "by hand" using level 3 BLAS kernels and a software environment like PVM or MPI. However, this would have made such an approach dependent on their existence and backward compatibility. Since PVM is already slowly disappearing, while imposing strict backward compatibility on MPI may be holding it to too high a standard, the decision not to follow this path seems

to be very good indeed. It has led in the first place to the development of the BLACS (Basic Linear Algebra Communication Subroutines), a package that defines portable and machine independent collection of communication subroutines for distributed memory linear algebra operations [45, 98]. The essential goals of BLACS are:

- simplifying message passing in order to reduce programming errors,
- providing data structures to simplify at the level of matrices and their subblocks,
- portability across a wide range of parallel computers, including all distributed memory parallel machines and heterogenous clusters.

In the BLACS, each process is treated as if it were a processor – it must exists for the lifetime of the BLACS run and its execution can affect other processes only through the use of message passing. Processes involved in the BLACS execution are organized in two-dimensional grids and each process is identified by its coordinates in a grid. For example, if a group of consists N_p processes then the grid will have P rows and Q columns, where $P \cdot Q = N_g \leq N_p$. A process can be referenced by its coordinates (p, q), where $0 \leq p < P$ and $0 \leq q < Q$.

The BLACS provides structured communication in a grid. Processes can communicate using the point-to-point paradigm or it is possible to organize communication (broadcasts) within a "scope" which can be a row or a column of a grid, or even the whole grid. Moreover, the performance of communication can be improved by indicating a particular hardware topology [45].

```
      integer iam, nprocs, cntx, nrow, ncol, myrow, mycol, i, j
      integer lrows, lcols
      real*8 a(3), h
*
      call blacs_pinfo(iam,nprocs)
      if (nprocs.eq.-1) then
        if (iam.eq.0) then
           print *,'How many processes ?'
           read *, nprocs, a(1), a(2)
        end if
      call blacs_setup(iam,nprocs)
      end if
```

```
* determine grid size
    lrows=int(sqrt(real(nprocs)))
    lcols=lrows
* init the grid
    call blacs_get(0,0,cntx)
    call blacs_gridinit(cntx,'C',lrows,lcols)
    call blacs_gridinfo(cntx,nrow,ncol,myrow,mycol)
* broadcast or receive
    if ((myrow.eq.0).and.(mycol.eq.0)) then
      call dgebs2d(cntx,'A',' ',2,1,a,4)
    else
      call dgebr2d(cntx,'A',' ',2,1,a,4,0,0)
    end if
*
    h=(a(2)-a(1))/real(nprocs)
    a(1)=a(1)+real(iam)*h
    a(2)=a(1)+h
    a(3)=0.5*h*(f(a(1))+f(a(2)))
    call dgsum2d(cntx,'A',' ',1,1,a(3),4,0,0)
    if ((myrow.eq.0).and.(mycol.eq.0)) then
      print *,'result is ',a(3)
    end if
    call blacs_barrier(cntx,'A')
    call blacs_exit(0)
```

This program is analogous to the program presented in Section 6.4.1. This time, however, the communication infrastructure is expressed in terms of calls to BLACS routines. This illustrates the fact that while geared toward linear algebra, the BLACS can be also utilized as a general set of communication routines. It should be also noted that several version of BLACS were implemented based on PVM, MPI and vendor-provided message passing routines.

The PBLAS (Parallel Basic Linear Algebra Communication Subprograms) [31] is a set of distributed vector-vector, matrix-vector and matrix-matrix operations (analogous to the sequential BLAS) with the aim of simplifying the parallelization of linear algebra programs. The basic idea of PBLAS is to distribute matrices among distributed processors (i.e. BLACS processes) and utilize BLACS as the communication infrastructure. The general class

of such distributions can be obtained by matrix partitioning like

$$A = \begin{pmatrix} A_{11} & \dots & A_{1m} \\ \vdots & & \vdots \\ A_{m1} & \dots & A_{mm} \end{pmatrix},$$

where each subblock A_{ij} is $n_b \times n_b$. These blocks are mapped to processes by assigning A_{ij} to the process whose coordinates in a grid are

$$((i-1)\text{mod}P, (j-1)\text{mod}Q).$$

Finally, ScaLAPACK is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD computers and networks of heterogeneous computers [21]. It provides the same functionality as LAPACK for workstations, vector supercomputers, and shared-memory parallel computers. As LAPACK was developed by utilizing calls to the BLAS routines, ScaLAPACK is based on calls to the BLACS and PBLAS kernels.

Summarizing, the current state of the art of both compiler-based and language-based parallelization is such that neither can be relied on when considering efficient implementation of parallel algorithms. In the best case, the optimizing compiler should be able to fine-tune the hand-parallelized code to match the low level parallelism available in the hardware and to match various detailed hardware parameters of a given high performance computer (e.g. processor characteristics, structure, sizes and latencies of various levels of cache memories, etc.). When considering parallelization of an existing code to be executed on one of the available parallel computers, the availability and popularity of environments supporting such a process should be selected. At present, the best choices seem to be OpenMP, MPI or a combination of them. Attention needs to be paid to the large and constantly growing number of existing libraries of modules out of which parallel programs can be assembled as well as to complete problem solving environments that can be applied to efficiently find the solution.

7 Concluding Remarks

A few predictions for the future of parallel scientific computing can be risked on the basis of the above summary of the state of the art in parallel computing. The development of computer architectures is clearly pointing out to the increasing importance of parallel computing. The newest processors, which are about to become the standard for workstations, e.g. the Itanium

architecture [5] from Intel or the Opteron and Athlon 64 architectures from AMD [1], involve a continuous increase of internal complexity (e.g. ever more sophisticated branch prediction), increased word size to 64 bits and introduce various forms of threading [5]. Each of these factors increases the total number of instructions that will be executed inside of the processor at any given time. While the available software tools (e.g. the optimizing compilers) are lagging behind the advances in computer hardware, their abilities are steadily improving and they should be able to support microparallelization successfully as well as to handle hierarchical memory latencies for a given architecture.

For smaller and medium size problems (where these notions are dynamic and their extensions change together with hardware capabilities), workstations with multiple processors and global shared memory become very popular. At the present time, dual-processor desktop computers have become so popular that their applicability to video processing and multimedia production has been thoroughly analyzed in the *AV Video Multimedia Producer* magazine [60]. This indicates clearly that parallel computing "has reached the masses." At the same time, on the high end of parallel computing, further substantial increase in computational power is about to take place. The three leading projects are: a 40 Tflop computer from Cray to be installed at Sandia National Laboratory in 2004 (Red Storm Project), the 100 Tflop ASCI Purple (consisting of 12544 processors) and the 131072 processor BlueGene/L computer capable of peak performance of 360 Tflops. The latter machines will be built by IBM and installed in 2005 in Lawrence Livermore National Laboratory [32].

In the context of scientific computing, parallelization can be viewed on multiple levels that are nicely illustrated by the dense linear algebra software discussed above. On the low level, highly optimized building blocks will continue to be developed (e.g. BLAS kernels) with optimization coming from the hardware vendors or from research projects such as the ATLAS project. These building blocks will be combined into software libraries (e.g. ScaLAPACK). They will be also utilized in the development of environments designed for the solution of a class of problems (e.g. eigensolvers for complex symmetric non-Hermitian matrices, parallel solutions to various classes of constraint optimization problems, large scale data mining problems, etc.).

For shared memory parallel computers, it can be expected that tools similar to OpenMP will remain a standard for software writing, while the language extensions and parallelism supporting languages like Java will take some time to reach the required level of efficiency for the solution of larger problems. Both these approaches will be used in building software libraries,

or will be combined with the use of software modules stored in various libraries to solve real-life applications. Due to the relative simplicity of the underlying architecture and a substantial body of knowledge about writing software for shared memory computers that has been accumulated over more than 20 years of their existence, it will be possible to achieve a high level of efficiency relatively easily.

The situation will be slightly more complicated in the case of distributed memory environments (clusters and top-of-the-line supercomputers). Here, the distributed computing model based on message passing is the most likely to remain the standard for software writing and tools like the MPI (which has evolved into MPI 2 [2]) are most likely to be used to support it. Outside of the simpler well-structured problems that are easily amenable to parallelization, and afford high levels of efficiency, a lot of work in performance tuning will be required. In some cases, this work may be sometimes avoided due to the fact that the wall-clock problem-solving time is the most important for the user. The users may accordingly be willing to forgo the extra effort in code tuning and instead add more/faster hardware thus achieving the required/acceptable solution time. However, it is very likely that the solution of extremely large problems on computers with multiple layers of latencies (in current supercomputers of the ASCI project at least seven levels of latency can be accounted for) will still be mostly done "by hand" with the solution being individually developed for each particular problem to match the underlying hardware architecture.

Another approach, which also follows the general reasoning that more hardware can be substituted for fine tuning and "local" efficiency, is the grid. While it is unclear whether this is the computing paradigm for the future (many think so and are likely to be correct, but the real applications have not materialized yet), clearly there exist classes of computational problems that even today can benefit from the grid-like architecture and the availability of unused computational power. Any problem that can be divided into a number of relatively small (computational time between a few hours and a day on the weakest machines that are a part of the grid) and completely independent tasks that also has a favorable ratio of computation to communication (a small amount of data is transferred across the grid while a large amount of computation is then applied to it) is a definite candidate for a grid-based solution. Due to the steadily increasing network bandwidth and growing amount of available unused computational power, a very large amount of research is devoted to this area (with large companies such as IBM becoming seriously financially involved in such efforts). It can be therefore expected, that while the fruits of the research will be available some time in

the future, this is and will remain for some time to come one of the hottest research areas in distributed computing. In this context, it is worth mentioning that in addition to the focus on the plain computational power and efficiency, the grid also involves attempts of adding intelligence to problem solving. Here it is assumed that large problems will have separate parts that may run best on different computers. For instance, a part of the problem may work well on a shared memory vector machine, while another part may be matched with a cluster computer. The environment is to recognize this and try to use this information to optimize the solution process. One of the more interesting projects in this group is the NetSOLVE environment [28, 29, 86].

While some of predictions presented here may not materialize, one thing is for certain, parallel and distributed computing is taking over the world of computing – it is here to stay and to grow.

References

[1] http://www.amd.com/us-en/Weblets/0,,7832_8366,00.html.

[2] http://www.mpi-forum.org/docs/docs.html.

[3] ACM Trasaction on Mathematical Software. http://www.acm.org/toms.

[4] The Beowulf Project. http://www.beowulf.org.

[5] *Intel Itanium Architecture Software Developer's Manual.* Intel.

[6] The netlib repository. http://www.netlib.org.

[7] *Supercomputer Prospectives – 4th Jerusalem Conference on Information Technology*, Maryland, May 1984. IEEE Computer Society Press.

[8] OpenMP C and C++ application program interface. http://www.openmp.org, October 1998.

[9] *Intel Architecture Optimization. Reference Manual.* Intel, 1999.

[10] OpenMP Fortran application program interface. http://www.openmp.org, November 2000.

[11] J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagner. *Fortran 95 Handbook.* MIT Press, 1997.

[12] S. J. Allan and R. R. Oldehoeft. HEP SISAL: Parallel functional programming. In J. S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, Scientific Computation Series, pages 123–150. MIT Press, Cambridge, MA, 1985.

[13] P. Alpatov, G. Baker, H. C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn. PLAPACK: Parallel linear algebra libraries design overview. In *Proceedings of Supercomputing'97 (CD-ROM)*, San Jose, CA, Nov. 1997. ACM SIGARCH and IEEE.

[14] P. R. Amestoy, M. Daydé, and I. S. Duff. Use of level 3 BLAS in the solution of full and sparse linear equations. In J.-L. Delhaye and E. Gelenbe, editors, *High Performance Computing: Proceedings of the International Symposium on High Performance Computing, Montpellier, France, 22–24 March, 1989*, pages 19–31, Amsterdam, The Netherlands, 1989. North-Holland.

[15] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostruchov, and D. Sorensen. *LAPACK User's Guide.* SIAM, Philadelphia, 1992.

[16] G. Baker, J. Gunnels, G. Morrow, B. Riviere, and R. van de Geijn. PLAPACK : High performance through high-level abstraction. In *Proceedings of the 1998 International Conference on Parallel Processing (ICPP '98)*, pages 414–423, Washington - Brussels - Tokyo, Aug. 1998. IEEE USA.

[17] I. Bar-On and M. Paprzycki. A parallel algorithm for solving the complex symmetric eigenproblem. In M. Heath et al., editors, *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, 1997. SIAM.

[18] I. Bar-On and M. Paprzycki. A fast solver for the complex symmetric eigenproblem. *Computer Assisted Mechanics and Engineering Sciences*, 5:85–92, 1998.

[19] I. Bar-On and M. Paprzycki. High performance solution of complex symmetric eigenproblem. *Numerical Algorithms*, 18:195–208, 1998.

[20] N. H. F. Beebe. EISPACK: numerical library for eigenvalue and eigenvector solutions. World-Wide Web document., 2001.

[21] L. Blackford et al. *ScaLAPACK User's Guide.* SIAM, Philadelphia, 1997.

[22] D. Bollman, F. Sanmiguel, and J. Seguel. Implementing FFTs in SISAL. In J. T. Feo, C. Frerking, and P. J. Miller, editors, *Proceedings of the Second Sisal User's Conference*, pages 59–65, Livermore, CA, 1992. LLNL. CONF-9210270.

[23] W. Brainerd, C. Goldbergs, and J. Adams. *Programmers Guide to Fortran 90*. McGraw-Hill, 1990.

[24] I. Brodsky. Scale grid computing down to size. *NetworkWorld*, January 27:47, 2003.

[25] J. Burt. Grid puts supercomputing at enterprises' fingertips. *eWeek*, January 20:32, 2003.

[26] D. C. Cann, J. Feo, and T. DeBoni. SISAL 1.2: High-performance applicative computing. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing (2nd SPDP'90)*, Dallas, 1990.

[27] B. Carpenter et al. Toward a java environment for SPMD programming. *Lectrue Notes in Computer Science*, 1470:659–668, 1998.

[28] H. Casanova and J. Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[29] H. Casanova and J. Dongarra. Applying NetSolve's network-enabled server. *IEEE Computational Science & Engineering*, 5(3):57–67, July/Sept. 1998.

[30] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, San Francisco, 2001.

[31] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. Whaley. LAPACK working note 100: A proposal for a set of parallel basic linear algebra subprograms. http://www.netlib.org/lapack/lawns, May 1995.

[32] B. A. Cipra. Sc2002: A terable time for supercomputing. *SIAM News*, 36(2), 2003.

[33] M. J. Daydé and I. S. Duff. The RISC BLAS: a blocked implementation of level 3 BLAS for RISC processors. *ACM Transactions on Mathematical Software*, 25(3):316–340, 1999.

[34] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, 1997.

[35] J. Dongarra. Performance of various computer using standard linear algebra software. http://www.netlib.org/benchmark/performance.ps.

[36] J. Dongarra, J. Bunsch, C. Moler, and G. Steward. *LINPACK User's Guide*. SIAM, Philadelphia, 1979.

[37] J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[38] J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14:1–17, 1988.

[39] J. Dongarra, I. Duff, D. Sorensen, and H. Van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, 1991.

[40] J. Dongarra, I. Duff, D. Sorensen, and H. Van der Vorst. *Numerical Linear Algebra for High Performance Computers*. SIAM, Philadelphia, 1998.

[41] J. Dongarra et al. *PVM: A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, 1994.

[42] J. Dongarra, F. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Rev.*, 26:91–112, 1984.

[43] J. Dongarra and L. Johnsson. Solving banded systems on parallel processor. *Parallel Computing*, 5:219–246, 1987.

[44] J. Dongarra and D. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37:73–83, 1995.

[45] J. J. Dongarra and R. C. Whaley. LAPACK working note 94: A user's guide to the BLACS v1.1. http://www.netlib.org/blacs, May 1997.

[46] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the Sparse Basic Linear Algebra Subprograms: The new standard from the BLAS Technical Forum. *ACM Transactions on Mathematical Software*, 28(2):239–267, June 2002.

[47] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale Sparse Matrix Package (YSMP) – I : The symmetric codes. *Int. J. Numer. Meth. in Eng.*, 18:1145–1151, 1982.

[48] R. Ekersand, K. Cullers, J. Billingham, and L. Scheffer. *A Roadmap for the Search for Extraterrestrial Intelligence*. SETI Press, 2002.

[49] A. Ferrari. JPVM: Network parallel computing in java. *Concurrency: Practice and Experience*, 10, 1998.

[50] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C–21:94, 1972.

[51] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[52] K. Gallivan, M. Heath, E. Ng, J. Ortega, B. Peyton, R. Plemmons, C. Romine, A. Sameh, and R. Voight. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, 1991.

[53] B. Garbow, J. Boyle, J. Dongarra, and C. Moler. *Matrix Eiigensystems Routines – EISPACK Guide Extension*. Lecture Notes in Computer Science. Springer-Verlag, New York, 1977.

[54] George, A. and Ng, E. SPARSPAK : Waterloo sparse matrix package "User's Guide" for SPARSPAK-B. Research Report CS-84-37, Dept. of Computer Science, Univ. of Waterloo, 1984.

[55] W. Gropp and E. Lusk. *A User's Guide for mpich, a Portable Implementation of MPI version 1.2.0*.

[56] J. R. Gurd. The manchester dataflow machine. In I. S. Duff and J. K. Reid, editors, *Vector and Parallel Processors in Computational Science*, pages 49–62. North-Holland, 1985. Computer Physics Communications 37 1-3 1985.

[57] J. Gustafson. Reevaluating Amdahl's law. *Comm. ACM*, 31:532–533, 1988.

[58] J. Gustafson, G. Montry, and R. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Stat. Comput.*, 9:609–638, 1988.

[59] F. G. Gustavson. New generalized data structures for matrices lead to a variety of high performance algorithms. *Lecture Notes in Computer Science*, 2328:418–436, 2002.

[60] F. G. Gustavson. Waiting for your chip to come in. *AV Video Multimedia Producer*, February:14, 16, 2003.

[61] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. *ACM SIGPLAN Notices*, 35(1):39–52, Jan. 2000.

[62] D. Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20:740–777, 1978.

[63] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 1996.

[64] P. Hochmuth. Gushing over linux. *NetworkWorld*, 4(7):46, 2003.

[65] R. Hockney and C. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger Ltd., Bristol, 1981.

[66] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartizs, E. A. Vavalis, K. Y. Wang, and S. Weerawarana. ELLPACK: A numerical simulation programming environment for parallel MIMD machines. In *Proceedings 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News*, pages 96–107, Sept. 1990. Published as Proceedings 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News, volume 18, number 3.

[67] C. F. Jr. Grid-dy determination. *NetworkWorld*, June 1:43, 46.

[68] C. Koelbel, D. Loveman, R. Schreiber, G. S. Jr, and M. Zosel. *The High Perhormance Fortran Handbook*. MIT Press, 1994.

[69] D. Kuck. *Structure of Computers and Computations*. Wiley, New York, 1978.

[70] S. Lakshmivarahan and S. K. Dhall. *Analysis and Design of Parallel Algorithms: Algebra and Matrix Problems*. McGraw-Hill, New York, 1990. $44.

[71] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Soft.*, 5:308–329, 1979.

[72] C. D. Marsan. Grid vendors target corporate applications. *Network-World*, January 27:26, 2003.

[73] M. Metcalf and J. Reid. *Fortran 90/95 Explained.* Oxford University Press, 1999.

[74] J. Modi. *Parallel Algorithms and Matrix Computation.* Oxford University Press, Oxford, 1988.

[75] M. Musgrove. Computers' shelf life gets livelier. *Washington Post*, December, 10:E01, 2002.

[76] J. Ortega and R. Voight. *Solution of Partial Differental Equations on Vector and Parallel Computers.* SIAM, Philadelphia, 1985.

[77] P. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, San Frncisco, 1996.

[78] M. Paprzycki. Parallel matrix multiplication - can we learn anything new? *CHPC Newsletter*, 7(4):55–59, 1992.

[79] M. Paprzycki and C. Cyphers. Gaussian elimination on cray y-mp. *CHPC Newsletter*, 6(6):77–82, 1991.

[80] M. Paprzycki and C. Cyphers. Multiplying matrices on the cray - practical considerations. *CHPC Newsletter*, 6(4):43–47, 1991.

[81] M. Paprzycki, H. Hope, and S. Petrova. Parallel performance of a direct elliptic solver. In M. Griebel et al., editors, *Large Scale Scientific Computations of Engineering and Environmental Problems*, pages 310–318, 1998.

[82] M. Paprzycki, I. Lirkov, and S. Margenov. Parallel solution of 2d elliptic pde's on silicon graphics supercomputers. In Y. Pan et al., editors, *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, pages 575–580. IASTED/ACTA Press, 1998.

[83] M. Paprzycki, I. Lirkov, S. Margenov, and R. Owens. A shared memory parallel implementation of block-circulant preconditioners. In

M. Griebel et al., editors, *Large Scale Scientific Computations of Engineering and Environmental Problems*, pages 319–327, 1998.

[84] M. Paprzycki and P. Stpiczyński. Parallel solution of linear recurrence systems. *Z. Angew. Math. Mech.*, 76(S2):5–8, 1996.

[85] M. Paprzycki and J. Zalewski. Parallel computing in Ada: An overview and critique. *Ada Letters*, 17:62–67, 1997.

[86] J. S. Plank, H. Casanova, M. Beck, and J. J. Dongarra. Deploying fault-tolerance and task migration with NetSolve. *Future Generation Computer Systems*, 15(5–6):745–755, Oct. 1999.

[87] S. F. Reddaway, G. Bowgen, and S. V. D. Berghe. High performance linear algebra on the AMT DAP 510. In G. Rodrigue, editor, *Proceedings of the 3rd Conference on Parallel Processing for Scientific Computing*, pages 45–49, Philadelphia, PA, USA, Dec. 1989. SIAM Publishers.

[88] K. Regan. Unsold gateway PCs to serve as on-demand grid network. http://www.techextreme.com/perl/story/20230.html.

[89] J. R. Rice. Ellpack 77 user's guide. Technical Report CSD–TR 226, Purdue University, West Lafayette, IN, 1978.

[90] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines : EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, 1976.

[91] P. Stpiczyński. A new message passing algorithm for solving linear recurrence systems. *Lecture Notes in Computer Science*, 2328:466–473, 2002.

[92] P. Stpiczyński and M. Paprzycki. Fully vectorized solver for linear recurrence systems with constant coefficients. In *Proceedings of VECPAR 2000 – 4th International Meeting on Vector and Parallel Processing, Porto, June 2000*, pages 541–551. Facultade de Engerharia do Universidade do Porto, 2000.

[93] D. K. Taft. Ibm bolsters grid computing line. *eWeek*, February 3:24, 2003.

[94] L. N. Trefethen and D. Bau. *Numerical Linear Algebra.* SIAM, Philadelphia, 1997.

[95] R. A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* Scientific and Engineering Computing. MIT Press, Cambridge, MA, 1997.

[96] R. A. van de Geijn and J. Overfelt. Advanced linear algebra object manipulation. In R. A. van de Geijn, editor, *Using PLAPACK: Parallel Linear Algebra Package*, Scientific and Engineering Computing, pages 42–57. MIT Press, Cambridge, MA, 1997. Chap. 3.

[97] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform.* SIAM, Philadelphia, 1992.

[98] R. C. Whaley. LAPACK working note 73: Basic linear communication algebra subprograms: Analysis and implementation across multiple parallel architectures. http://www.netlib.org/blacs, June 1994.

[99] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27:3–35, 2001.

[100] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison–Wesley, 1996.

[101] H. Zima. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

[102] Z. Zlatev, J. Wasniewski, and K. Schaumburg. *Y12M solution of large and sparse systems of linear algebraic equations: documentation of subroutines*, volume 121 of *Lecture Notes in Computer Science.* Springer-Verlag Inc., New York, NY, USA, 1981.