Introduction to Parallel Computing

A.G. Hoekstra
Section Computational Science
University of Amsterdam
alfons@wins.uva.nl

Table of Contents

1 ..... Preface 3

2 ..... Basic Concepts 4

2.1 ..... General introduction 4

2.2 ..... Parallel Processing Hardware 19

2.3 ..... Paradigms of Parallelism 21

2.4 ..... Conditions for Parallelism 27

2.5 ..... Parallelization 31

2.6 ..... Message Passing 39

3 ..... Part II : A Case Study, the Guitar String 52

3.1 ..... Introduction 52

3.2 ..... The model 52

3.3 ..... The Solver 55

3.4 ..... Parallelization 57

3.5 ..... Performance Analysis 60

4 ..... Part III : Performance Analysis and Benchmarking 62

4.1 ..... Introduction 62

4.2 ..... Performance Evaluation Metrics 62

4.3 ..... Time Complexity Models 67

4.4 ..... Benchmarking 78

5 ..... To Conclude 85

6 ..... References 85

1 Preface

This syllabus accompanies the project "Parallel Rekenen en Visualisatie", or, in English, Parallel Computing and Visualization. This syllabus relates to the "Introduction to Parallel Computing" part of the project

Introduction to Parallel Computing is course in the Bachelor Computer Science at the University of Amsterdam in the department of Computer Science. To enter the course students should have a basic knowledge of computer architecture and calculus, and should have experience in programming in C.

I hope that you enjoy this course as much as I on my first acquaintance with the topic of parallel computing. Besides its eminent importance in Computer Science, it's just a lot of fun to get these things going, and try to squeeze as must power out of a large distributed or parallel computer as possible.

Alfons Hoekstra
Amsterdam, May 2005

2 Basic Concepts

2.1 General introduction

2.1.1 Some history

Computer Science is an enormously rapid developing field. Yesterday, just a few people used the internet, today we have the success story of the Web and people in the computing field discuss computational grids as 'the next internet', and nobody knows which revolution is lurking behind the doors to bring us tomorrow's breakthroughs. History in Computer Science therefore is a very strange matter. You would probably consider the time before the second world ware as prehistoric, the 1940's until the 1970's as the Middle Ages, and with the start of the PC revolution the 1980's and 1990's as the Renaissance for Computer Science. And indeed, the 1980's and 1990's have been a Renaissance in the field of computing. Old concepts, which emerged during the early developments in electronic computers, or even before that time, have actually been realized in hardware. Decrease in switching speed of digital circuits, VLSI technique allowing to integrate a micro computer on a single chip, fast memories and I/O devices, but especially the introduction of parallelism on every level of digital computing has dramatically increased computational speeds. Today, in the modern times of Computer Science (or should I say post-modern already?) parallelism and parallel computing is abundant, and is adopted by many fields in Science and Engineering as the standard way to achieve required computational speeds. Just 10 years ago this was not so. In this lecture we will discuss the basics of parallel computing, allowing you to understand in detail all the issues at stake in modern grid computing applications.

In only half a century the computational speed has moved up from 100 floating point operations (flop) per second on the EDSAC (summation of a large array of real numbers [1]) to 36 Tflop/s on the NEC Earth-Simulator, a massively parallel computer containing 5120 powerfull NEC vector processors [2]. In other words, this means an increase in computational speed with a factor of 10^{11}. The increase in clock speed of the processors itself was not that dramatic, 2 μs on the EDSAC1 [3] to approximately 1 ns on top of the bill super computers, i.e. only a factor of 2 million. Furthermore, if we account for the fact that the arithmetic itself is much more accurate (36 bit fixed point arithmetic compared to 64 bit floating point arithmetic), it is obvious that the largest gain in computational speed has to be attributed to architectural - and (system) software innovations. In this course we will discuss many of these innovations.

Computational Speed

The computational speed of a computer is usually expressed in floating point operations per second. So, if your computer is able to perform say, 10 million floating point operations, or flop, per second, we would say that the speed is 10 Mflop/s. The 'M' means mega, or 10^6. The other big numbers are:

K	kilo	10^3
M	mega	10^6
G	giga	10^9
T	tera	10^{12}
P	peta	10^{15}

It is also handy to know the small numbers, here they are:

m	milli	10^{-3}
μ	micro	10^{-6}
n	nano	10^{-9}
p	pico	10^{-12}
f	femto	10^{-15}

Low level parallelism has played a major role in these innovations. Bit-parallel arithmetic operations, functional units for e.g. addition or multiplication operating in parallel, and pipelining inside those functional units, allowing overlapping operations on several data items all had a major impact on the performance of processors, and you will find many of such ideas in modern RISC processors. Introduction of banked main memories and fast caches between main memory and registers of a processor dramatically increased the transfer rate of data and instructions between memory and registers. Finally, the important innovation of vector registers and vector operations, introduced in commercial machines by Seymour Cray, opened the way to come close to the theoretical peak performance of the inverse of the clock cycle time (of course on well-tuned problems).

To go beyond the impressive performance figures of single processors, the next logical step is to let more than one such powerful processor work on a problem in concert. Now we need to make a clear distinction between the *inter CPU parallelism* described in the

previous paragraph, i.e. which is immersed inside one complete general purpose processor unit, and parallelism which is introduced by means of a replication of complete processor units. By *Parallel Computing* we exclusively mean computing using hardware consisting of replicated processor units. These processor units may contain every level of sophisticated inter CPU parallelism, but may also consist of very simple bit-serial processors. In this course we restrict ourselves to Parallel Computing. The inter CPU parallelism is part of courses on computer architecture, and is shortly reviewed in the part "Introduction to Parallel Architecture". Also, Refs [4, 5, 6] are very good books on all these topics.

Probably the first reference to parallel computing was by L. F. Manebrea. In his 1842 publication "*Sketch of the Analytical Engine Invented by Charles Babbage*" he writes (we take this quotation from Ref [7], page 8): "*...Likewise, when a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes...*". A very early description of a parallel computer is by L. F. Richardson in 1922 (this example is taken from Ref. [8], the original is Ref. [9]). In his book *Weather Prediction by Numerical Process* we find in chapter 11.2: "*...If the time-step were 3 hours, then 32 individuals could just compute two points so as to keep pace with the weather, if we allow nothing for the very great gain in speed which is invariably noticed when a complicated operation is divided up into simpler parts, upon which individuals specialize. If the co-ordinate checker were 200 km square in plan, there would be 3200 columns on the complete map of the globe. In the tropics the weather is often foreknown, so that we may say 2000 active columns. So that 32×2000=64,000 computers would be needed to race the weather for the whole globe. That is a staggering figure. Perhaps in some years time it may be possible to report a simplification of the process. But in any case, the organization indicated is a central forecast-factory for the whole globe, or for portions extending to boundaries where the weather is steady, with individual computers specializing on the separate equations. Let us hope for their sake that they are moved from time to time to new operations....*". Of course, Richardson's computers were *people*.

Richard Feynman, the famous Nobel price winning physicist, presents an example of an early, actually working computational pipeline. In his book *Surely You're Joking Mr. Feynman* he talks about how he spent his time in Los Alamos, during the second world war, and how his group would carry out numerical calculations (Feynman headed the so-called IBM group). They had figured out that they needed a number of IBM machines (in that time IBM built mechanical tabulators, multipliers etc.), and they arranged them in a loop, thus 'programming' their mechanical computer. The idea was to put a punch card in the first machine, do a calculation, and put the resulting card in the next machine. Feynman's problem was that this was not a very fast way to work. They needed nine months to figure out three problems. From this point I will cite from Feynman's book, where he describes their trick to speed things up (see part 5, Los Alamos from below): "*But one of the secret ways we did our problems was this. The problems consisted of a bunch of cards that had to go through a cycle. First add, then multiply-and so it went through the cycle of machines in this room, slowly, as it went around and around. So we figured a way to put a different colored set of cards through a cycle too, but out of phase. We'd do two or three problems at a time.*" This trick however presented Feynman with another problem. At some time he had to come up with the answer of a specific problem within a month; "*...So Bob Christy came down and said, "We would like the results for how this thing is going to work in one month"-or some very short time, like three weeks. I said, "It's impossible". He said, "Look, you're putting out nearly two problems a month. It takes only two weeks per problem, or three weeks per problem." I said, "I know. It really takes much longer to do the problem, but we're doing them in parallel. As they go through, it takes a long time and there's no way to make it go around faster.*" Bob Christy was impressed by the amount of problems that Feynman and his group could deliver every month, but he had no idea of the real cycle time of the problems, as they went through the mechanical computer. This is typical of what is called pipelined parallelism. In the following chapters we will discuss this in more detail and you will be able to analyze the speed of delivering answers of problems as a function of the number of

computers in the pipeline and the number of problems you are actually running through the pipeline. However, you could off course try such analysis yourself right know, and later on see how it matches with the analysis in this syllabus.

The first electronic digital computer, the ENIAC, contained much internal parallelism (e.g. 20 accumulators) and could in principle be programmed in a multi program way (different micro programs for each accumulator). However, this design was too ambitious for its time, and the first stored-program computers, the EDVAC, the EDSAC and the UNIVAC1, which are true von Neumann type sequential computers, were much more powerful. These sequential architectures slowly evaluated into today's powerful RISC processor and vector processors.

During the first years of electronic computing in the 1950's, much research was devoted to parallel computing. Unfortunately, these ideas could not be realized into efficient machines until the 1980's. Here, efficient should be read as faster or cheaper than more conventional sequential architectures. Based on a theoretical paper by von Neumann in 1952 [10], Unger proposed a practical design for a two dimensional array of processors [11]. This line of development resulted in the ILLIAC IV and the ICL DAP [see e.g.12, 13], which were arrays of processors, each with their own memory.

The idea to assemble a large number of processors, each executing their own program and working on their own data, into one parallel computer can be traced back to a paper in 1959 by Holland [14]. This paper influenced later work by Pease, who introduced the concept of the hypercube architecture [15], which resulted in the very successful cosmic cube [16]. A detailed historical account of parallel computing can be found in reference [7], chapter 1.

The second half of the 1980's shows a true explosion of parallel computing, with the introduction of many successful massively parallel systems. Examples are the CM-2 of Thinking Machines, the Intel hypercubes (iPSC 1 and 2), the Meiko Computing Surface, and the Parsytec Super Cluster. The last two are based on the transputer, a (European) microprocessor that was specifically designed for parallel computing. We may view these systems as the first generation parallel computers. The experience gained with these first generation systems has proven that massively parallel systems can be build and reliably operated, and that parallel programs can run very efficiently on hundreds or thousands of processors. Most of these systems however could not compete with the vector supercomputers. In the beginning of the 1990's some new massively parallel systems entered the market. These second-generation systems, such as the CM-5, or the Intel Paragon have comparable, or even better performance figures as e.g. the Cray YMP which was the most powerful vector supercomputer at that time. Today massively parallel computers completely dominate the top end computing. You may encounter vector processors as a single node of a massively parallel computer, but the fastest computers in use today are all based on the concept of massively parallel computing architectures.

Supercomputers

The speed of a computer is an important property. And like Formula 1 racing cars, the field of computing also has its little niche where top speeds and trying to go faster and faster is the goal. In section 2.1.2 you will see that many applications require ever-faster computers. By the end of the 1970's Semour Cray launched the vector processor. Using very sophisticated technology, the vector computers became the Formula 1 racing cars, the fastest computers around. They where called supercomputers and dominated the arena of very fast computing for more than 15 years. Only the last few years we see a clear shift from using 'traditional' vector supercom-puters to massively parallel computers

We have not at all talked about software. Without disregarding the many important breakthroughs in e.g. system software, high level programming languages, and programming environments, programmability of parallel systems has been, and still is, a serious drawback of parallel computing. Although the parallel programming environments of modern parallel systems alleviate much of the problems encountered in programming parallel computers, it is generally acknowledged that programming in parallel, and porting large sequential codes to parallel systems is not a trivial task. One paradigm of parallel computing, the Single Program Multiple Data

paradigm of parallel computing [17], emerged as a very useful way of parallel programming for large scientific and engineering applications, and most modern parallel systems support this mode of programming. This paradigm will be discussed in much more detail in later sections.

2.1.2 The Need for Speed

Computers get faster and faster every day. We all know this, by the time you have installed your new PC at home, Intel just announced a next generation of again faster chips. Why is this? Off course, Intel wants to make a lot of money, but more important, the chase to ever increasing computational speeds is fuelled by a strong application pull. Multimedia, graphics, web applications, advanced text processing, games or business applications just need an enormous amount of computer speed.

The same is true, but even more in the extreme, in the field of technical computing, information management, and embedded systems. Researchers and engineers constantly need more computational power in order to carry out larger or more accurate simulations, or just to run their simulations continuously faster (e.g. in weather forecasting). In Figure 1 a number of applications that are known to require a lot of computational speed are arranged according to their required computational speed and memory. The fastest processors available today (order of 1 Gflop/s peak speed) are in the lower speed regions of Figure 1. The same is true for typical main memory that is present in workstations of top-of-the-line PC's. It is clear that most of the applications in Figure 1 require computer resources far beyond a single workstation or PC.

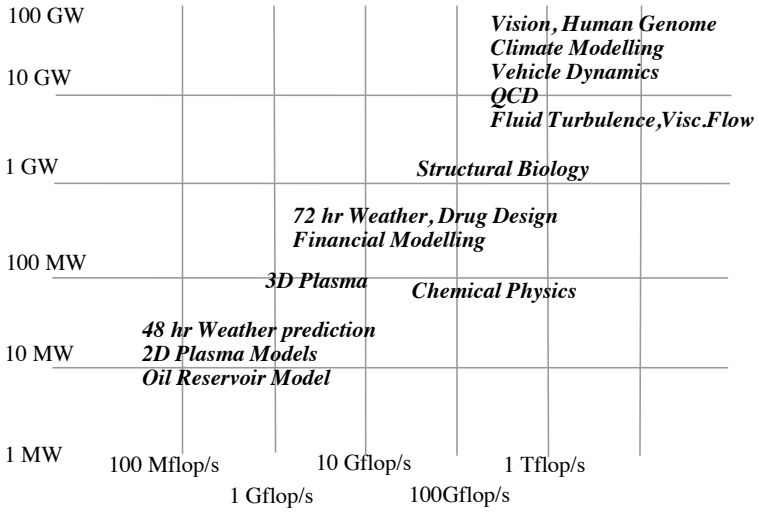


Figure 1: The computational speed (on the x-axis) and memory requirements (on the y-axis) of a number of grand challenge applications. The 'W' stands for Word, meaning a Word in memory, that could be e.g. 32 or 64 bits long.

In fact, Figure 1 is based on a figure taken from a report [18] that was written in 1992 by a commission headed by Al Gore, later Vice-President of the USA under Bill Clinton. In 1992 even the fastest supercomputers were hardly able to deliver a speed of more than 100 Mflop/s. Yet, the applications, which were coined *Grand Challenges* because of their strategic value and high relevance for the nation's well-being, require up Tflop/s performance or more and main memories of 100 Gbyte. All this was the starting point of an enormous research and engineering project in the field of *High Performance Computing* (HPC). The ultimate goal was to build computers able to deliver the Tflop/s. Only six years later, in 1998, this holy grail of HPC was obtained. Not very surprising, the race continues. Applications still need more power, and currently research groups are aiming at the next barrier, the Pflop/s.

2.1.3 Examples of Grand Challenge Applications

Here we show two examples of grand challenge applications. These examples were taken from recent results of European HPC projects.

We first consider an example of simulation of airflow around an airplane. Computation of the airflow around plane shapes has become an important issue for the airplane industry. Access to numerical tools allows a reduction in design time and real tests. The ultimate aim of simulation is to allow faster and reliable design of airplanes. This means that the numerical tool must be faster in terms of speed. This is the reason for intensive optimization and parallelization work. Reliable means that the results obtained with the numerical tool must be more and more accurate. This is the reason for the development of new models and the implementation of new numerical methods. The combination of fast and more reliable results in the need for HPC systems. Figure 2 shows an example of a computation of flow around an airplane.

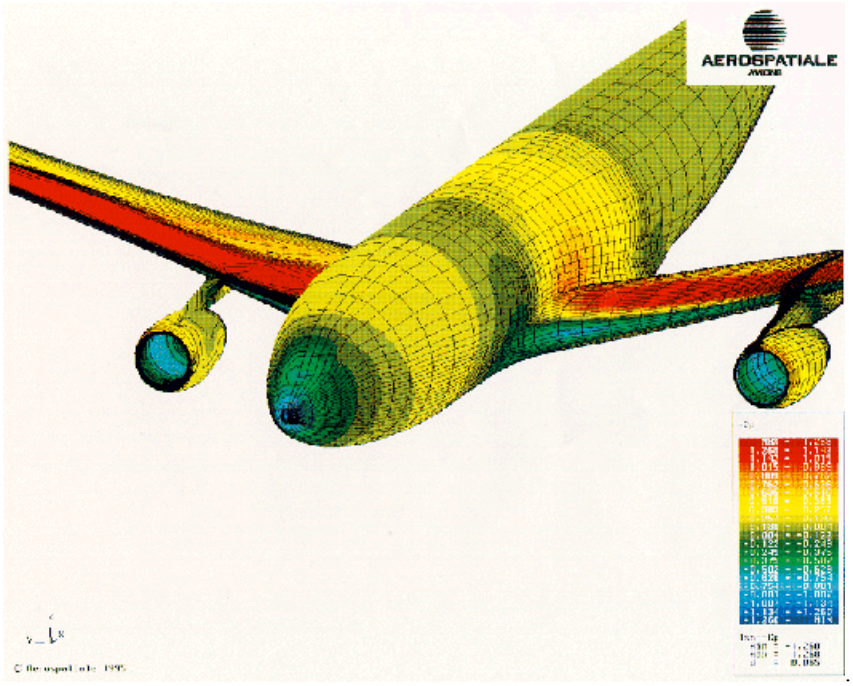


Figure 2: Example of computation of flow around an airplane.

Another example comes from medicine. New computing technology is allowing heart specialists to diagnose defects more effectively. The use of HPC computing technology has dramatically reduced the time it takes to create a 3-D model of the heart from which accurate measurements can be made. This new capability will help improve diagnosis of heart disease, which is likely to affect half of the population in industrialized countries. A team of specialists based in Greece has developed a system, which reconstructs a 3D computerized model of the heart from a sequence of 2D X-ray images. The use of powerful computers delivers the model in just a few minutes, which allows more cost effective use of equipment and allows consultants to use their time more effectively. Accurate measurements of characteristics such as the volume of the heart chambers and motion of the heart walls can be determined using the system. Figure 3 shows an example of heart models.

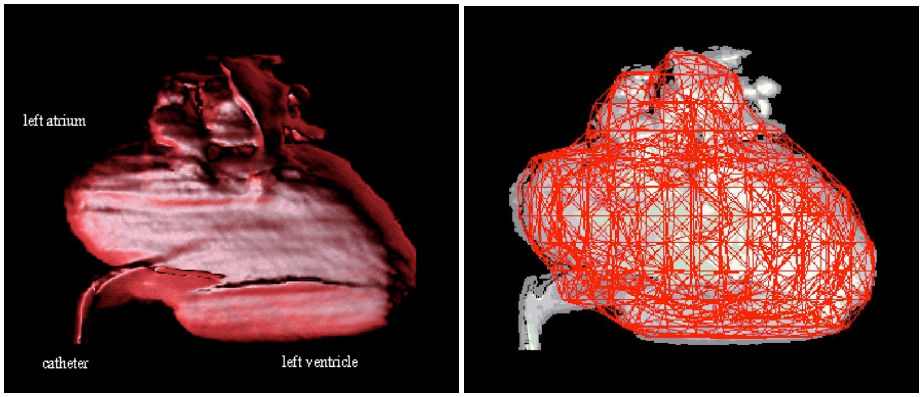


Figure 3: Example of 3D modeling of the heart.

Other typical examples include e.g. car crash analysis, high quality animation technology (think of Hollywood blockbusters like Jurassic park), but also Data Mining, i.e. complicated queries in very large databases.

2.1.4 Why Parallel?

So far we heard a lot about Grand Challenge Applications, the Need for Speed, and HPC, but it is not immediately clear why it is necessary to go for parallel computing. As was already mentioned in section 2.1.1, parallel programming is more difficult than normal sequential programming. Therefore, staying away from parallelism may not be such a bad idea. However, as was also suggested in section 2.1.1 and 2.1.2, single processor performance may just not be enough to reach the enormous computing speeds required by many applications (see e.g. Figure 1).

Let us first go through a nice academic exercise, and try to assess fundamental limitations on the computational speed of one single processor. As was argued in section 2.1.1, well-tuned architectures are now capable to deliver one floating point operation during every clock cycle on well tuned problems. Clearly, this is a theoretical upper limit on the computational speed, but it suits our goal. Define $R_{\max}$ as the maximum computational speed and δ as the clock cycle time. We can immediately write

$$R_{\max} = \delta^{-1} \ . \qquad [1]$$

In the past several arguments, based on the finite speed of light and Heisenberg's uncertainty principle, have been proposed to find theoretical upper limits for $R_{\max}$ (see e.g. Ref. [19]). We will now investigate if these two physical constraints really pose a theoretical upper limit to the maximum calculation speed. According to Heisenberg the accuracy of a time measurement Δt and an energy measurement ΔE are related by $\Delta E\ \Delta t > h/2\pi$, h is Planck's constant. If we identify δ with Δt, we find $\delta > h/(2\pi\ \Delta E)$ and $R_{\max} < 2\pi\ \Delta E/h$. Let the energy difference between a 0 bit and a 1 bit be equal to ΔE. Strictly speaking the uncertainty principle does not impose an upper limit to R_{max}, because ΔE can be increased. This would however imply an increasing dissipation of heat in the processor, which in turn will be the limiting factor. However, because h is very small, small values for ΔE already give a very large upper bound. For instance, if we put $\Delta E = 2$ eV (band gap in a semiconductor, energy of a visible photon) we already find $R_{\max} < 3$ Pflop/s. We can conclude that the uncertainty principle does not impose any limits on attainable computational speed (yet).

The finite speed of light however will present a true constraint on computational speed. Computation requires transportation of information from memory to the processing unit and back. Assume that for every floating point operation information must travel a total distance d. The cycle time is now limited by the finite speed of light c according to $\delta > d/c$, and consequently $R_{\max} < c/d$. To move information over a distance of 1 cm requires a minimum of $0.01 / 3\times10^8 = 0.033$ ns (remember, 'n' was nano, i.e. 10^{-9}). For an addition of

two numbers we move two pieces of information from memory to processor, and one piece of information (the result of the addition) back again. Assume that the distance between processor and memory is 1 cm. This means that for the minimal computing time we find $\delta > 0.1$ ns and $R_{max} < 10$ Gflop/s. We have to reduce d to 300 μm to have a $R_{max} < 1$ Tflop/s. Even if memory (e.g. a large cache) and processing unit are integrated on a single chip, such a small value of d is probably not possible. Therefore we can conclude that the finite speed of light presents an upper limit to processing speed of say 10 to 100 Gflop/s. So the grand challenge problems cannot be simulated on these “ultimate” serial computers, and parallelism must be introduced.

Despite the previous arguments, the real limitation at this moment is technology. Switching times are in the order of 1 ns, and it is very difficult to reduce this in case of CMOS technology. In other words, for CMOS technology $R_{max} \sim 1$ Gflop/s. Another real limitation is the memory bandwidth that cannot cope with the increased processor performance. Although all kinds of caching techniques are employed [see e.g. 6], many applications suffer from a memory bottleneck that severely reduces the computational speed that can be obtained with a single processor.

A possible candidate to reduce switching speeds is super-conducting logic based on Josephson junctions [20]. The first attempts, made in the 1970's, were not very successful. Although IBM demonstrated a small signal processor with a cycle time of 665 ps, they terminated their efforts for a super-conducting computer in 1983. In Japan, research in super conducting chips, as part of the project 'High Speed Computing Systems for Scientific and Technological Use' [21], continued during the 1980's. In 1990 a Fujitsu group demonstrated a working chip containing 23,000 Josephson junctions, capable of performing a multiplication of two 8 bit numbers in 240 ps and addition of two 13 bit numbers in 410 ps [22]. Currently, we see a "third wave" of research interest in super-conducting logic [23], based on the so-called rapid single flux quantum (RSFQ) logic [24]. Simple systems, based on RFSQ logic, with switching times of 20 ps have been demonstrated. However, to scale these systems to complete processors, and to build (super-conducting) memories with fast enough access times remains a huge technological challenge [20,23].

To end this discussion on novel technology we should mention the revolutionary concepts of complete optical computers [see e.g. 25], true quantum computers [see e.g. 26], or even DNA computers [27]. It is difficult to foresee whether these ideas will result in working devices that can compete with existing technology. However, it is true that currently a lot or research is directed in these exiting fields, and history has taught us that we should never try to look in the crystal ball with respect to computer architecture.

We must however conclude that massively parallelism is currently the only way to satisfy the computational requirements of many relevant (grand challenge) applications.

So far we only discussed a very strong application pull for parallel computing. This is not the full story. Clearly, not everybody needs the enormous performance of top of the bill parallel computers. Much computational research relies on powerful workstations. In this segment we observe a strong technology push, where cost effectiveness plays an important role. Parallel systems, with computational speeds comparable to a Cray, can be built for a fraction of the cost of a Cray by using off-the-shelves components (e.g. PC motherboard, fast ethernet cards, public domain software such as Linux, etc.). Therefore, many institutes acquire or even build themselves such a “poor man's supercomputer”, and in this way turn to parallel computing. The so-called Beowulf class of computers is an example [28]. Furthermore, the trend of *cluster computing* or *distributed computing*, i.e. turning a network of workstations into a loosely coupled parallel computer also provides a relatively easy and cheap first step into parallel computing.

2.1.5 High Performance Computing

High Performance Computing (HPC) is the field of computer science that aims to develop computing systems (i.e. computer hardware, system software, run time support systems, tools) and applications in order to obtain high performance, in terms of computing speeds

and turn around times, by relying on parallel computing. A high performance computer system is in general a distributed heterogeneous computer structure, which may consist of workstations, supercomputers, massively parallel computers, I/O devices, all connected via fast networking. An example of a HPC system is shown in Figure 4.

It consists of a local area network of workstations, two massively parallel computers (the Cray T3E and the IBM SP2) and a parallel vector supercomputer (the NEC SX-4). Furthermore, it has a CAVE, a large Virtual Reality system. Some fast network connects all these systems. A main challenge is to operate such a heterogeneous distributed HPC system as a single *metacomputer*. A user of a metacomputer would not need to have any knowledge of the underlying system architecture, he would just log on to the metacomputer and submit jobs to it without even knowing where the job is executed and where exactly his data is stored. The metacomputer would schedule the job to the most appropriate computer. All this is absolutely not trivial and the concept of metacomputing is part of active research in the HPC community. These ideas now turned into the exiting field of grid computing.

All components in Figure 4 can also be considered as an HPC system. Let us now make a distinction between *massively parallel computers* and *distributed computers*. Massively parallel computers usually contain many tightly coupled processors in a single machine. The processors are connected through some highly optimized interconnection network. The Cray T3E and IBM SP2 are examples of massively parallel computers. A distributed computer on the other hand is a collection of, in general heterogeneous computers (PC's, workstations) that are connected via a local area network. For instance, a network of workstations, such as the one in your faculty, is a distributed computer if we let the workstations execute a parallel program.

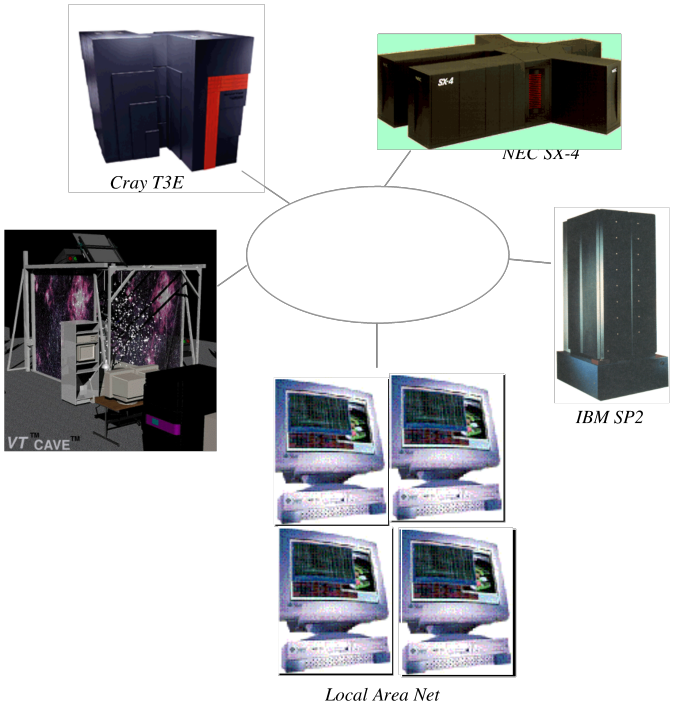


Figure 4: An example of a HPC system

In the beginning of the 1990's the distinction between massively parallel computers and distributed computers was quite large. A massively parallel computer would typically contain specially designed processors and interconnection networks, whereas a distributed computer would contain e.g. Sun workstation connected through Ethernet. Today, things have changed dramatically. The microprocessors that are used in PC's (so, the Intel

Pentium), Macintosh (the Motorola PowerPC) or workstations (the DEC Alpha or Sun UltraSparc) are very fast and usually are relatively cheap. Currently, almost all successful parallel computers are based on processors that are also encountered in workstations or PC's. For instance, the ASCI-RED computer contains Intel Pentium processors, and the Cray T3E contains DEC Alpha processors. So, the main difference between distributed and parallel computers nowadays seems to be the interconnection network and system software. However, the distinction between the two is blurring. From later sections you will understand the main differences between the two, which are mainly due to communication latency and throughput.

2.1.6 Some Parallel Computers

Table 1 provides an overview of current vendors of parallel computers and some of the systems they are selling. This information is based on a yearly overview of parallel computing systems that is compiled by A.J. van der Steen and J.J. Dongarra [29]. This table is based on data of July 2001.

The only European vendor is Quadrics. All other European manufactures of parallel computers (Parsytec, Meiko, Alenis, BBN, etc) no long participate on this market or just don't exist anymore. Fujitsu, Hitachi, and Nec are Japanese, and the rest are US companies. In market share the US companies are by far the biggest. Almost all parallel systems are based on commodity processors, such as the PowerPC, the Alpha, the SPARC or the MIPS processors. Usually, parallel computers have a few ten's to a few hundreds of those processors. Only the fastest computers on earth have a thousand to ten thousand processors.

Actually, in the past larger parallel computers, in terms of number of processors, have been built. The first massively parallel computer to beat the vector supercomputers was the Connection Machine CM-2. In its largest configuration, this machine had 64,000 processors. However, these processors were relatively simple, bit-serial processors.

Vendor	**type**	**Processor**	**remarks**
Compaq	AlphaServer SC	Alpha 21264a (EV67)	6-512 processors
Cray Inc.	SV1ex.	CMOS based vector processor	8-18 processors
	T3E.	Alpha 21164	6-2176 processers
Fujitsu	AP3000	Sun Sparc	4-1024 processors
Fujitsu	VPP5000	Vector processor combined with RISC processor	4-128 processors
Fujitsu/Siemens	PRIMEPOWER	Fujitsy version Sun Sparc, the SPARC 64 GP	8-128 processors
Hitatchi	SR8000	PowerPC	4-512 processors
HP	9000 SuperDome	PA-RISC 8600	16-64 processors
IBM	RS/6000 SP	PowerPC or POWER3	8-2048 processors
NEC	SX5	proprietary vector processor	8-512 processors
	Cenju-4	MIPS R10000	8-1024 processors
Quadrics	Apemille	MAD chip	8-2048 processors
SGI	Origin 3000	MIPS R14000	4-512 processors
SUN	E10000 Starfire	UltraSPARC	16-64 processors

Table 1: An overview of vendors of parallel computers and their products, based on information of July 2001 [29]

Another very useful overview in the field of HPC is the Top 500 [2], a list of the 500 most powerful computers on earth. Table 2 shows the top 10 of November 2004. The R_{max} is measured using the so-called LinPeak benchmark. In the LinPeak benchmark one solves a

linear system of N equations with N unknowns using Gaussian elimination. In the original LinPack benchmark $N = 100$ or $N = 1000$. In LinPeak however, one is allowed to adjust N so as to reach a maximal performance. Usually this means that N is increased as much as possible. In section 4 the science of benchmarking is discussed in more detail, and you will understand why increasing the problem size N results in a higher performance. Again, for those readers who like to do a little puzzle, try this analysis yourself and compare it later with that in section 4.

The fastest computer is the IBM Blue Gene from the Department of Energy in the USA, with a performance of 70,7 Tflop/s on the LinPeak benchmark.

Manufacturer Computer	**Installation Site Location / Year**	**# Proc.**	R_{max} Gflop/s
1 IBM Blue Gene	IBM / DOE Rochester, USA / 2004	32768	70720
2 SGI Columbia	NASA Ames Research Center Mountain View, USA /2004	10160	51870
3 NEC Earth Simulator	Earth Simulator Center Yokohama, Japan / 2002	5120	35860
4 IBM MareNostrum	Barcelona Supercomputing Center Spain / 2004	3564	20530
5 California Digital Coorporatin Thunder	Lawrence Livermore National Laboratory Livermore, USA / 2004	4096	19940
6 HP ASCI Q	Los Alamos National Laboratory Los Alamos, USA / 2002	8192	13880
7 *selfmade* SystemX	Virginia Tech USA / 2004	2200	12250
8 IBM Blue Gene	IBM Rochester, USA / 2004	8192	11680
9 IBM	Naval Oceanographic Office USA / 2004	2944	10310
10 Dell Tungsten	NCSA Poughkeepsie, USA / 2003	2500	9819

Table 2: The fist 10 computers in the Top 500 list of June 2002.

Other very large computers are also USA government. Note the dominance of IBM in the top 10 of the fastest computers. IBM entered the parallel computing arena relatively late. However, after the very popular SP2 and SP3 series, the current BlueGene system is going to be a very big succes. Finally not number 7, a 'homemade' system, in the spirit of the Beowulf type of computers.

The Netherlands also has a number of entries in the TOP-500 list, see Table 3.

Position in list	Manufacturer Computer	Installation Site Location / Year	# Proc.	R_{max} Gflop/s
153	SGI Altix	SARA, Stichting Academisch Rekencentrum, Amsterdam / 2000	416	1793
164	IBM Xseries Cluster	Shell Rijswijk / 2004	512	1706
424	HP Integrity Superdome	Atos Origin / 2004	192	969
426	HP Integrity Superdome	HP Financial Services / 2004	192	969
458	IBM eServer	Rijks Universiteit Groningen	400	931

Table 3: Dutch entries in the Top 500 list of November 2004.

It is very interesting to perform some statistics on the TOP-500 list. For instance, Figure 5 shows the development of the performance of the TOP-500 computers. The fastest computer in 1993 had a LinPeak speed of 60 Gflop/s, and only 4 years later the fastest computer had a speed of 1 Tflop/s. This was the first time that a general purpose computer hit the Tflop/s boundary. That computer was the Asci Red, which even became faster in later months. Another interesting feature in this graph is the almost linear dependence of the total speed of all 500 computers in list as a function of time. In Figure 5 the speed is plotted on a logarithmic scale. This means therefore that the total speed of the TOP-500 computers grows exponentially with time. We may suspect that this behavior is directly related to the famous "Moore's law" which roughly states that the number of features on a chip doubles in approximately 18 months.

Another interesting statistic is the type of architecture that are installed. We distinguish between Massively Parallel Processors (MPP), Symmetric Multi Processors (SMP), Single Instruction Multiple Data (SIMD), Single Processor, Constellations and Clusters. The single processors are the "traditional" vector supercomputers, MPP and SMP are roughly speaking distributed – and shared memory parallel computers (including shared memory vector processors). In section 2.2.1 this is explained in some more detail. Figure 6 shows the amount of these systems in the TOP-500 list as a function of time. The most interesting conclusion is that the traditional single vector processor architecture has almost completely disappeared from the list, that the MPP architecture currently is the favored architecture for the fastest computers on earth. Moreover note that the SMP architecture is slowly disappearing from the list. Finally, the constellation is getting more and popular.

Figure 5: Performance development of the TOP-500 computers, until November 2005. We show the fastest (N=1) and the slowest (N=500) computer in the list, and the total performance of all computers in the list (picture taken from www.top500.org).

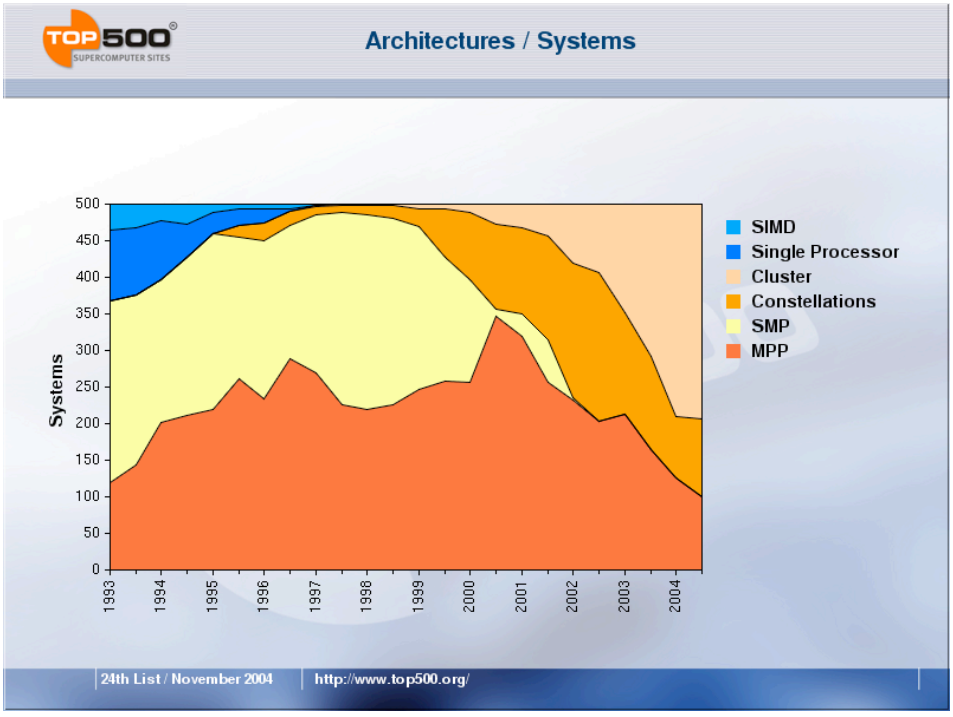


Figure 6: The number of Massively Parallel Processor systems (MPP), Single Instruction Multiple Data systems (SIMD), Symmetric Multi Processor systems (SMP),Single Processor systems, clusters, and constellations in the TOP-500 over the years until November 2004 (picture taken from www.top500.org).

The different types of CPU technology in the TOP-500 systems also show an interesting trend, see Figure 7. Clearly the vector processor is slowly being replaced by scalar processors as the main building block for parallel systems. Morever, a closer look at the

TOP-500 list shows that ECL technology is disappearing. More interestingly is the trend to use CMOS of the shelf technology. With this we mean that more and more parallel computers are built using standard microprocessors that are also used in single processor PC's or workstations. The most popular processors for parallel computers currently are the DEC Alpha, UltraSparc, MIPS R1000, and Intel Pentium family. Only a very small fraction still uses proprietary, i.e. specifically designed, CMOS technology.

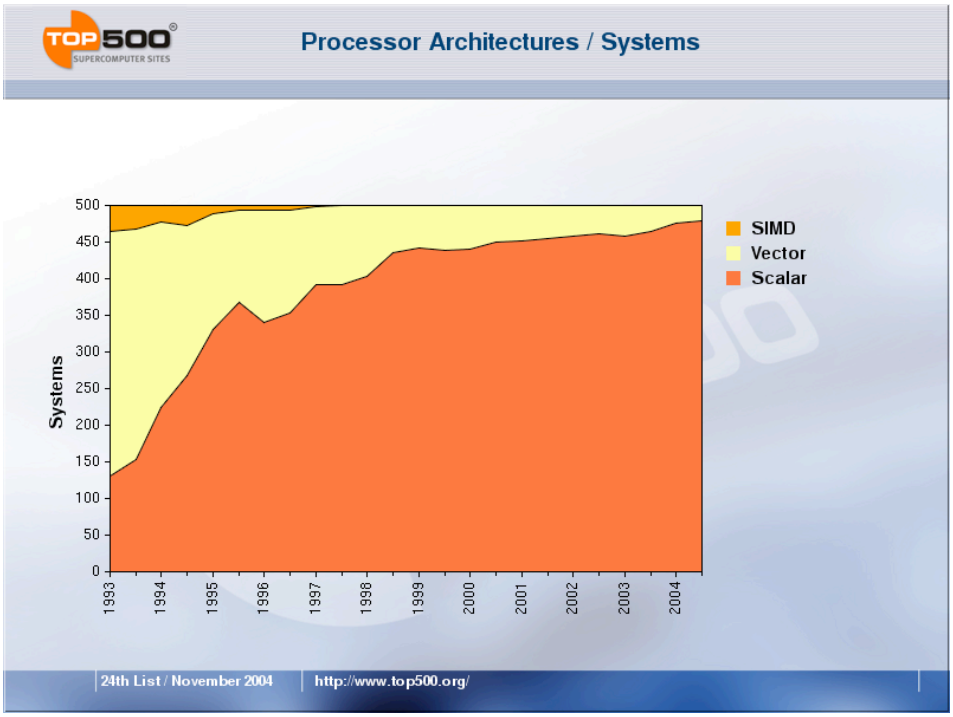


Figure 7: CPU technology used in TOP-500 computers over the years until November 2004 (Picture was taken from www.top500.org).

The TOP-500 also gives some nice data with respect to market share of manufacturers of HPC systems. Figure 8 shows the market share of manufacturers in the TOP-500. SGI/Cray inc. used to dominate this market. However the last years Sun, IBM and HP systems rapidly gain in popularity.

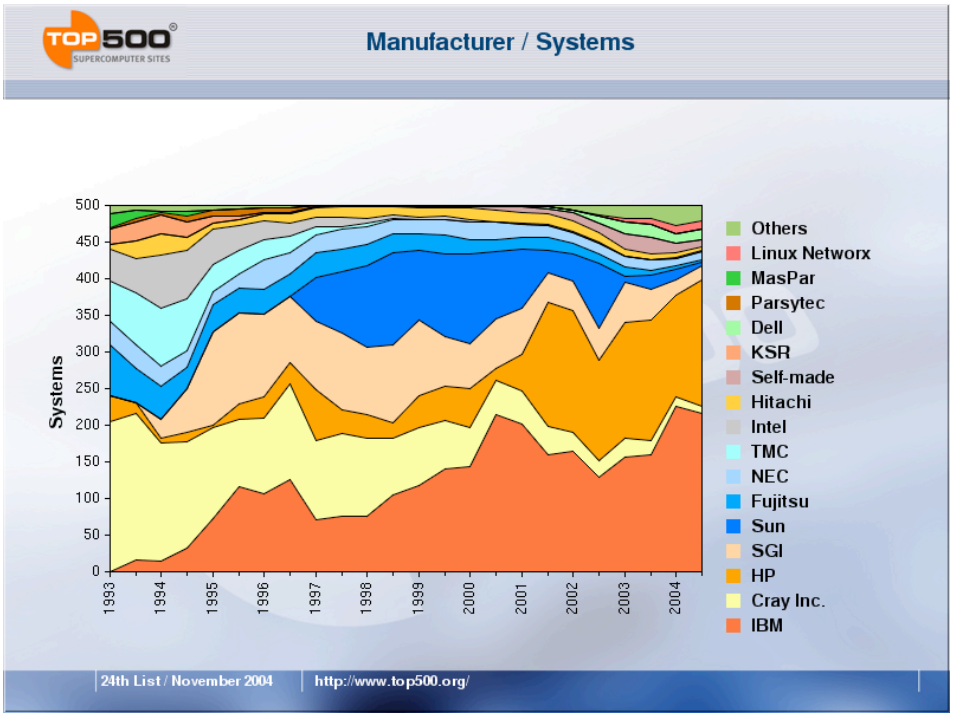


Figure 8: Market share of manufacturers in the TOP-500 over the years until November 2004 (Picture was taken from www.top500.org).

Finally, the TOP-500 tells us something about the types of applications for which the HPC systems are used. Figure 9 shows the application areas of the TOP-500 computers. Typically, some 5 % is classified (usually US defense applications) and 5 % are systems installed at vendor's sites. In first years of the list the share of academic, research, and industrial systems were more or less equall (25 to 30 %). Currently we see that half of the number of systems is installed at industrial sites, whereas research and academic sites both have a 20% share. From this we may conclude that parallel computing has become a mature main stream activity. We could present the statistics in another way, i.e. not just counting the number of machines, but the accumulated processing power installed at e.g. industry or academic and research institutes. In that case, in 2002, industry would amount for 24 %, academic institutions for 19 % and research institutions for 48 %. One should realize here that the top 10 computers are typically installed at research institutions.

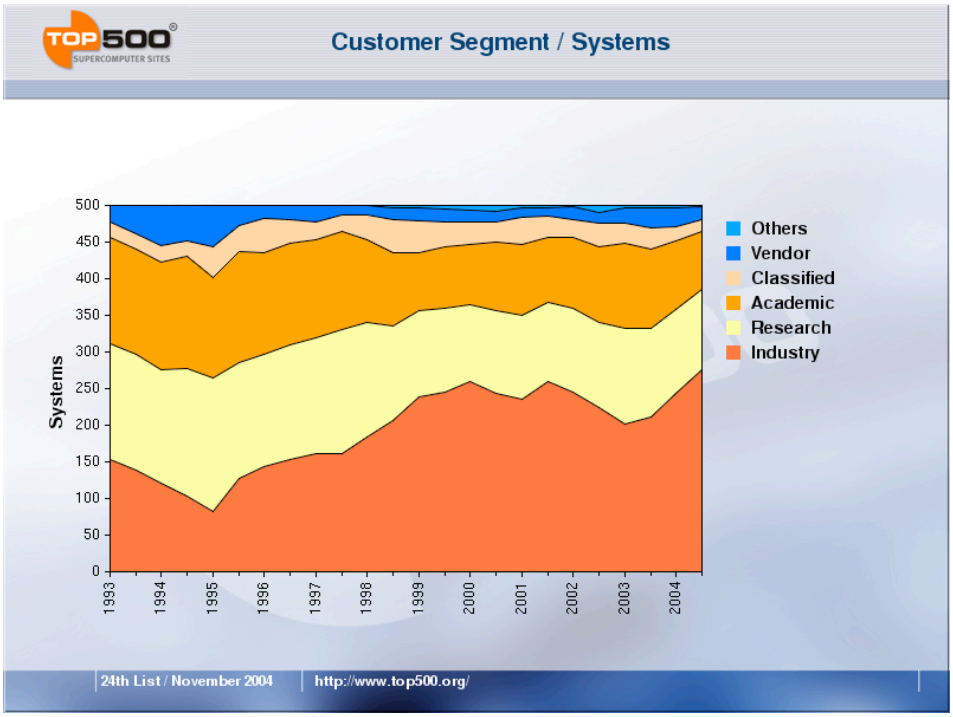


Figure 9: Application areas of the TOP-500 computers over the years.

2.1.7 Status and Prospects

High Performance Computing and more specifically parallel computing is here to stay. In 1994 Fox, Williams, and Messina write "Parallel Computing Works !" [30], a very large collection of applications of parallel computing, and indeed as such a collection of success stories of parallel computing. Parallel Computing is an accepted paradigm, not only in the academic and research community, but also in industry and commerce HPC is applied more and more. This does not mean that HPC is a completely established field. On the contrary, within computer science many people perform research in the field of HPC. The recent "Parallel and Distributed Computing Handbook" [31] provides a good overview of what is of interest in the community.

The main observation is that still no sufficient computer power and storage is available, resulting in parallel computing in order to get as much power as possible. This immediately induces new problems. How can you parallelize an application such that it runs efficiently on say hundreds to thousands of computers. How to do this in such a way that the performance of the parallel program stays at a high level if the number of processors is increased. This is the highly important question of *scalability*. In the next section we will investigate all this in more detail.

The development of parallel solvers and algorithms has come a long way. For many applications we know how to parallelize them. However, many "hard problems" exist which are extremely difficult to parallelize. Furthermore, a good parallelization also depends on the target parallel machine. It is important to develop virtual machine models. Finally, we have the *mapping* problem, i.e. how to map a parallel program as efficiently as possible to a (virtual) parallel machine. This mapping problem is known to be NP-complete.

Programming massively parallel computers remains a difficult task. Unlike what you are used to in sequential programming, standard tools like debuggers and profilers are still hardly available for parallel programmers. Another thing is that an enormous body of, in many cases "dusty deck Fortran" programs for many applications have been developed over the last 30 years. Migration of these programs to parallel systems is an expensive and time-consuming task. Despite much sponsoring by e.g. the European Commission, third

party software is still very slowly migrated to parallel systems. However, we should not be too pessimistic because as was shown in section 2.1.3, many examples of parallelization of such applications exist.

Not only parallelization, but also parallel programming paradigms are a new issue. In this lecture we concentrate on the by now well-established paradigm of *Single-Program-Multiple-Data* (SPMD) and parallel programs with explicit message-passing. We will introduce the current standard for SPMD programming, the *Message Passing Interface* (MPI). Other paradigms are Data-Parallel and Data-flow. You may wonder why, instead of an application programmer, not have the compiler do the parallelization, i.e. automatic parallelization as a kind of optimization flag for the compiler. Despite many claims by researchers, this approach has not been successful, except maybe for some synthetic example programs.

Other new issues are I/O and heterogeneity. Once you have a parallel application running at hundreds of processors, how to get the probably enormous data volumes in - and out of the parallel computer. The question of parallel I/O and out-of-core computing in parallel and distributed computers is still not answered satisfactory. An HPC system can contain many different processors or different operating systems. How to cope with this heterogeneity, for instance when sending data from one system to another, or when moving jobs from one system to another.

2.1.8 Our Agenda

We will not be able to touch upon all those issues in HPC. The main goal of this lecture is to provide an introduction in parallel computing. On some occasions we will introduce more advanced topics. We will mainly concentrate on

- Parallelism, which kinds of parallelism exist, and how can we discover parallelism;
- Parallelization, how to organize computations in such a way that we can exploit parallelism;
- Parallel programming, how to develop a correct and portable parallel program, using explicit message passing;
- Performance, what is the speedup compared to sequential computing and what is the scalability.

At this point it is nice to quote Hoare's law on parallel computing: "Inside every large program there is a small program struggling to get out". What Hoare meant, and you will probably also know this from your lectures on software engineering, is that in many applications and especially in technical computing a very small fraction of a computer code is responsible for the largest part of the execution time. It is clear that parallelization will first of all concentrate on this small program fraction.

2.2 Parallel Processing Hardware

2.2.1 Shared - versus Distributed Memory Architectures

Although the accompanying lecture "Introduction to Parallel Architecture" provides a much more in-depth overview of parallel processing hardware, we will, for the purpose of completeness, present the most basic taxonomy in which we can split up existing parallel processing hardware.

The first main characteristic of a parallel computer is its memory organization (see Figure 10). All processors may be connected to one single memory. This is called a *Shared Memory* (SM) architecture. Examples of SM architectures in Table 1 are e.g. the Sun starfire, or the SGI Origin. Also the traditional vector supercomputers are usually moderately parallel SM architectures. For instance, the Cray C90 at SARA in Amsterdam has 12 powerful vector processors on a large shared memory. The shared memory immediately provides each processor with a shared address space. This means that each processor can access all data that is available in memory. This results in a relative easy

programming of shared memory parallel computers. A disadvantage of shared memory architecture is the bad hardware scalability. The processors are connected to shared memory by some bus architecture. If the amount of processors increases, the amount of requests to the bus also increases, which may lead to bus contention. Therefore, massively parallel computers, containing hundreds of processors, are usually not shared memory architectures.

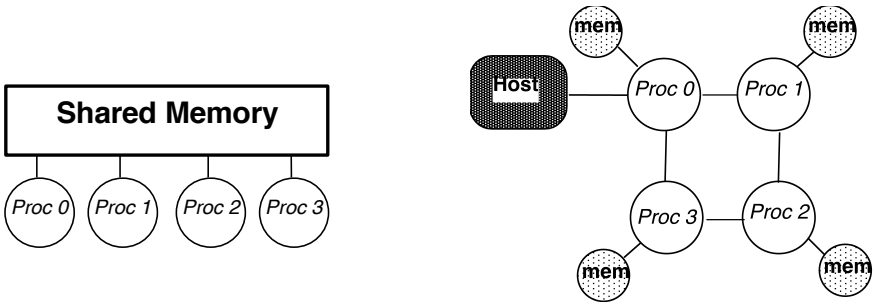


Figure 10: Shared memory (left) and distributed memory architecture (right).

Another possibility (see Figure 10) is that each processor has its own memory, which is not directly accessible for other processors. The processors have their own private memory and are connected through some network. This is the *Distributed Memory* (DM) architecture. Examples of DM architectures in Table 1 are e.g. the Cray T3E or the IBM SP3. Also a network of workstations can be considered as a DM architecture. This architecture is in principle scalable to thousands of processors. However, programming becomes more difficult. If a processor, say Proc 0 in Figure 10 needs a data item that resided in memory of Proc 2, some kind of message passing needs to be established between Proc 0 and Proc 2. Usually the programmer needs to set up explicit message passing. This seems like a bad thing, but current practice in parallel programming is indeed this explicit message passing programming.

A few more details are important for DM architectures. First, in Figure 10 a host was drawn. Early systems had a host, usually a workstation. Programs were developed and compiled on the host, and the parallel jobs were subsequently loaded on the parallel nodes. In modern DM systems the host is usually absent. A user directly enters a node of the parallel computer and from that node starts parallel jobs. The details depend strongly on the specific machine.

The early DM architectures were so-called Single Instruction machines. That means that all the parallel nodes execute, completely synchronously, the same program. The Thinking Machine CM-2, which was mentioned in section 2.1.6, is an example of such an architecture. Off course, it is also possible that each node in the DM architecture runs completely different programs, i.e. so-called Multiple Instruction machines. All modern DM architectures are of this type. The Single Instruction architecture was very popular until the beginning of the 1990's. Today most massively parallel systems are Multiple Instruction DM machines.

2.2.2 A Taxonomy

The large number of different architectures, and organization of processor networks and memories in parallel computers calls for a unifying taxonomy. Such taxonomy should provide just enough details for the users to distinguish the main characteristics of the system. A well-known and very useful taxonomy is due to Flynn [32]. This taxonomy distinguishes computers by their processing of data. A computer can perform *single* or *multiple* instructions, which can work on *single* or *multiple* data items. These concepts are combined into four main classes. The Single Instruction Single Data class (SISD) are the serial, von Neumann, computers. The Multiple Instruction Single Data class (MISD) is void. The Single Instruction Multiple Data class of computers (SIMD) issues a single stream of instructions, which operate on multiple data items. These multiple data items can be data items distributed among the processors of a Single Instruction DM

architecture. Therefore, the SIMD class contains the DM processors, which operate in single instruction mode, such as the ICL DAP or the CM-2. The last class, the Multiple Instruction Multiple Data type (MIMD) contains all multi computers. Therefore, this class contains both the SM architectures such as the Cray C90, and massively parallel DM systems like the Cray T3E or IBM SP2, but also clusters of workstations.

Flynn's taxonomy is very useful, but too broad to distinguish quite different architectures like the CrayC90 and the IBM SP2. Another often-made division is between shared memory and distributed memory computers. Raine [33] has used this as a starting point for a taxonomy which is based on the physical location of data in the computer, and how the data is presented to the programmer in "address space". The physical memory in a (parallel) computer can be shared or distributed, and the logical address space can be shared or disjoint. The Shared Address space Shared Memory class (SASM) contains the serial von Neumann computers, but also the vector supercomputers like the the Cray C90. The Disjoint Address space Shared Memory class (DASM) is void. The class of Disjoint Address space Distributed Memory computers (DADM) contains most massively parallel systems. Finally, the Shared Address space Distributed Memory class (SADM), also known as *Virtual Shared Memory* systems, is very interesting because it combines the potential of massively parallelism with the ease of programming (virtual) shared memory multiprocessors.

Combining both Taxonomies results in a reasonable distinction between broad classes of parallel computers. Try this yourself and see what happens.

2.3 Paradigms of Parallelism

2.3.1 Introduction

Parallelism has many different faces. I think that you all have an intuitive idea of the concept of parallelism and parallel computing, something like performing many operations or calculations at the same time. Try to image that you are a computer engineer and you were given a large bag of money and the order to do something with parallelism in computer architecture. What would you do? How would the designers from decades ago have thought about this intuitive concept of parallelism and transformed it into the success stories of today? I want to challenge the reader again, before continuing this section, think a while about different possible paradigms of parallelism. Try to formalize your intuitive ideas about parallelism and then continue reading.

Parallel computations can be divided in two main classes by looking how the computations are coordinated. Parallel computations are *synchronous* when all parallel operations are performed at the same time, under the control of a single clock. At each tick of a clock all parallel hardware units perform an operation. The other alternative is *asynchronous* parallel computation. Here all parallel units can operate completely independently of each other. If any coordination is necessary, because one parallel computation has to wait for the results of another computation, locks on data items are used. A semaphore is an example of such a lock.

Another important distinction in parallel computations is the *grain size*. Loosely defined grain size is a measure of the amount of computations in a single parallel task. In later sections this will be defined more precisely, but for now this loose definition suits our purpose. We can now distinguish between *fine grained* and *coarse grained* parallel computations.

Combining both concepts allows us to define four broad classes of parallelism, as in Figure 11. The synchronous fine-grained paradigm of parallelism results in the vector or array computer, and the synchronous coarse-grained paradigm is the SIMD class. For asynchronous parallelism, the fine-grained paradigm is the data flow type of computers and the coarse-grained paradigm is the MIMD class. Finally, an important paradigm of parallelism is Single Program Multiple Data (SPMD) which is in fact a hybrid between the strict coarse-grained synchronous and asynchronous paradigms. In the next sections all paradigms will be clarified. Note that these paradigms are not exhaustive. More

approaches are possible (e.g. data parallel, object orientation) but they are beyond the scope of this lecture.

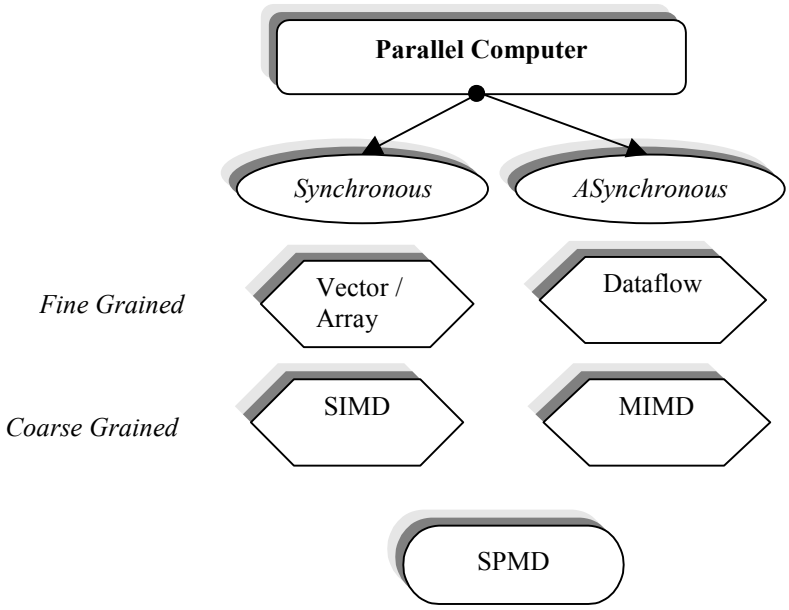
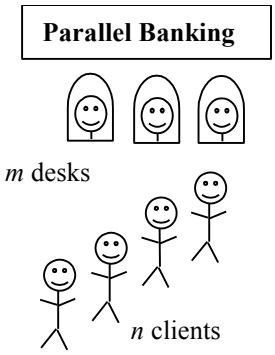


Figure 11: Four classes of parallel computers or parallel computations. SIMD is Single Instruction Multiple Data, MIMD is Multiple Instruction Multiple Data, and SPMD is Single Program Multiple Data.

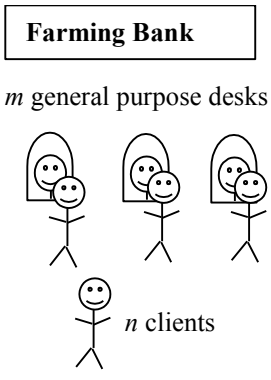


The previous sections we talked about parallel computer hardware. Now we discuss paradigms of parallelism that could be executed on any of these architectures. You will see that there is a natural mapping between the paradigms and hardware, although this mapping is certainly not one-to-one.

In order to clarify the paradigms of parallelism it is helpful to introduce an analogy. Assume a bank where n clients need to be served. Also assume that the bank has m desks and that each client i has a serving time of t_i. We are interested in the total serving T for all n clients. At time $t = 0$ all n clients enter the waiting room of the bank, and at time $t = T$ the last client leaves the bank again. Our bank started as a quite a small firm with only 1 desk, i.e. $m = 1$. In this case the total serving time is

$$T_1 = \sum_{i=1}^{n} t_i \,, \qquad [2]$$

where the subscript "1" denotes that serving time is for 1 processor, i.e. for 1 desk. The following sections will introduce several paradigms for parallelism and, using our banking analogy, analyze their merit.



2.3.2 Embarrassingly Parallel

First consider the case where we have fully independent, parallel processes. All these processes can in principle be different. This is the easiest case (but with a snag hiding somewhere….) and it is called *embarrassingly parallel*. Another phrase that is used is *farming* or *farming parallelism*. This is an example of fully asynchronous parallelism, where all processor units serve parallel jobs completely independent of each other. In fact, this is a normal

situation in a bank, where independent clients enter and are served asynchronously by the parallel general-purpose tellers.

Now, what about the serving time T. First assume the simplest case that the amount of clients is equal to the amount of desks, i.e. $n = m$, and that all serving times t_i are equal, i.e. for all i, $t_i = \tau$. Under these assumptions the serving time using m desks $T_m = \tau$, and the serving time if only one desk was used is $T_1 = m\tau$. Parallelization really helps here, the speedup S, defined as the serving time using 1 desk divided by the serving time using m desks, turns our to be

$$S = \frac{T_1}{T_m} = \frac{m\tau}{\tau} = m\,. \qquad [3]$$

This is the most ideal situation, using m processors (desks) results in a speedup of m, i.e. the original sequential execution time is reduced by a factor of m. This is what we hope to get from parallel computing. However, this is typically something you will never get. In all practical situations the speedup on m processors is usually smaller than m. An important topic, which is also treated extensively in this course, is to understand why the speedup is less than its ideal value and how the speedup can be maximized and maintained at a high value if the number of processors is increased.

Now, what about this hiding snag that I mentioned above. Here is a first source that reduces the speedup from its maximum value, *load imbalance*. Still assume that the amount of clients and desks is equal, but no longer assume that the serving times of the clients are equal. In that case

$$T_m = Max(t_1, t_2, \cdots, t_m) \qquad [4]$$

and $S < m$. In this situation some desks have already finalized serving their client, while others are still at work. In this way some desks are idle, which means, in processor terminology, that "cycles are lost". The amount of work is not equally divided among processors, which we call load imbalance, leading to a situation where some processors are idle while others are still processing. The result is that the speedup is lower than its ideal value.

The situation becomes more complicated when the amount of clients is larger than the amount of desks. Furthermore, the serving time of each client is usually not known in advance. This means that in farming parallelism, in order to have a good load balancing, some kind of intelligent scheduler must keep the processors busy. This is usually implemented as a Master/Slave model, where a master processor sends work to slave processors.

Consider the following example where we assume to have $m = 2$ desks and $n = 6$ clients. The service times for each client are shown in Table 4. In this case $T_1 = 20$. The question is how the divide the clients among the desks, such that we achieve an optimal speedup. Three different approaches are shown in Figure 12.

	Client 1	Client 2	Client 3	Client 4	Client 5	Client 6
time	6	1	5	3	3	2

Table 4: The service times of the six clients.

The first way is to naively send clients to desks in a round-robin way, i.e. client 1 to desk 1, client 2 to desk 2, client 3 to desk 1, etc. This results in total serving times for desk 1 of 14 and for desk 2 of 6, i.e. the parallel serving time for 2 desks becomes $T_2 = 14$, and the speedup in this case becomes $S = 20/14 = 1.43$. Clearly, this is not the most optimal way to divide the work. Better is to divide the work on a request basis. If a desk is ready, a next client is assigned to it. Assuming that assigning of clients can be done in zero time, we observe the following behavior. At the start, client 1 is assigned to desk 1, and client 2 to desk 2. After a time of 1 unit, client 2 is ready. Desk 2 requests for another client, and

client 3 is assigned to desk 2, etc. The resulting total serving time for desk 1 now is 11 and for desk 2 it is 9. The resulting speedup is $S = 11/20 = 1{,}82$. As we know the serving times in advance, we can also construct an optimal solution, which is shown in the right of Figure 12. Now $T_2 = 10$ and the speedup is the optimal of 2. Usually, the serving times are not known in advance, or the amount of jobs is so very large that construction of an optimal work distribution is very hard. In those cases, the work load distribution is performed using the "work-request" algorithm.

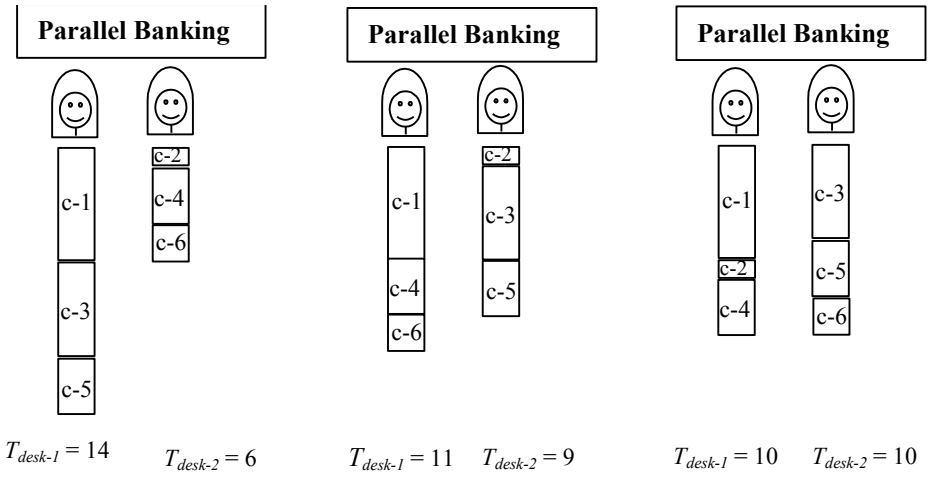
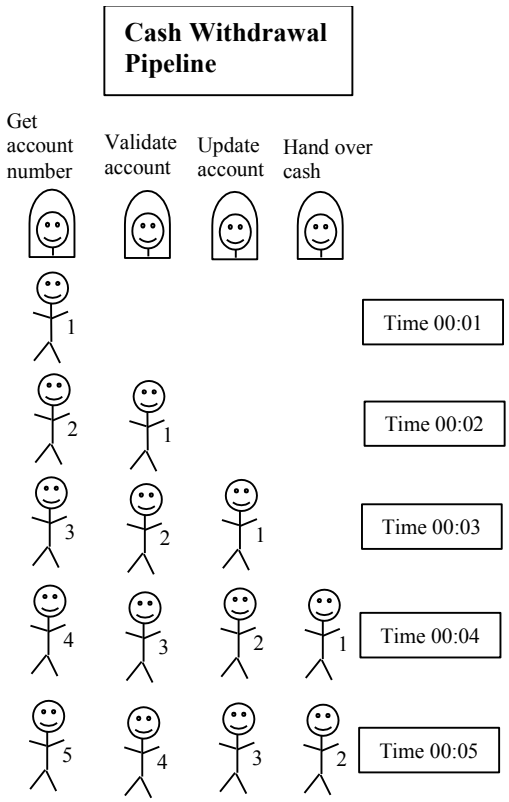


Figure 12: Dividing clients among available desks; c-1 is client 1, etc. The serving times are as in Table 4. The height of the boxes is proportional to the serving times. The left figure applies to naively dividing the work in a round-robin way, the middle figure is for the "work request" method and the right figure is the optimal solution.

2.3.3 The Vector Paradigm

Let us now move to the case of fine-grained synchronous parallelism. It is a bit tricky to have a nice analogy in our bank here, but let us assume that many clients have the same order, e.g. a cash withdrawal. The management of the bank decides to create specialized desks for this task. Even more, the management decides that each desk should specialize on one small subtask, e.g., a desk to get the account number of the client, a desk to validate the account, a desk to update the account, and a desk to hand over the required cash. Furthermore, all desks must work synchronously, and serve one client at the tick of a central clock. The clients move from one desk to the next. Note that this procedure is the same as the mechanical pipeline of Richard Feynman in section 2.1.1. This architecture is called a vector unit, or a vector pipeline. In modern RISC architecture such pipelines are e.g. present for addition or multiplication. Also, multiple pipelines may be present. Typically, the vector paradigm is an example of low level parallelism, which is present within a single processor unit.



At time 00:00 the pipeline is empty. At time 00:01 the first client is served at the first desk. At time 00:02 the first client is served at the second desk, while the second client enters and is served at the first desk. This process continues, and after 4 time steps the first client is done and leaves the bank. Now, after every clock tick a client is done and

leaves the bank. In other words, the computational pipeline is first filled, after which a result is produced at every clock cycle.

What do we gain with this trick? Suppose that we have a pipeline with m desks, or stages and suppose that a tick of the clock takes a time τ. Without the pipeline, the serving time $T = mn\tau$, because each client requires a time $m\tau$ and we have n clients. With the m-stage computational pipeline in place the serving time becomes $T_p = (m + n - 1)\tau$, because we need m clock ticks to fill the pipeline and then n more clock ticks for each item to leave the pipeline. In this way the speedup becomes

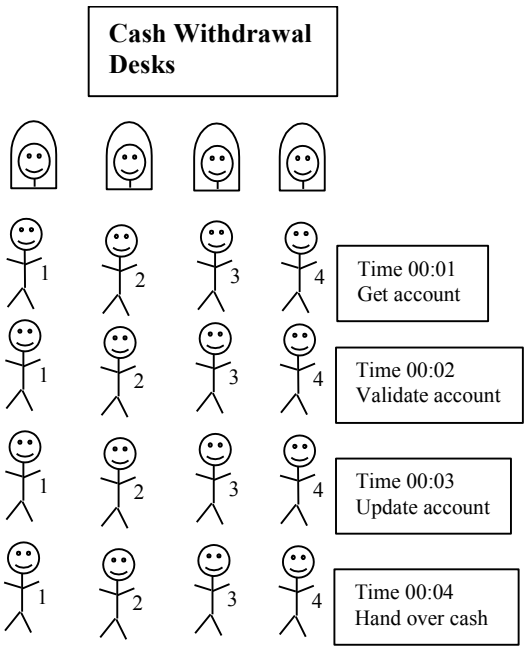
$$S = \frac{T}{T_v} = \frac{mn}{m+n-1}, \qquad [5]$$

which, in the limit for many clients, i.e. large n, becomes

$$\lim_{n\to\infty} S = \lim_{n\to\infty} \frac{mn}{m+n-1} = m. \qquad [6]$$

This means that for a large amount of clients, or for large vector lengths, the speedup that can be obtained in a computational pipeline is again close to the ideal speedup.

In combination with very small clock ticks, the vector paradigm laid the foundation for the vector supercomputers that were mentioned in previous sections. Especially in technical and scientific computing, were many regular computations on large vectors and matrices are necessary, this paradigm of parallelism is very successful. It is clear why the vector paradigm is fine-grained. It relies on breaking op computations into a usually small set of sub operations that are chained into a pipeline. The sub operations themselves are quite limited. We now move to the next synchronous paradigm, the coarse-grained SIMD.



2.3.4 The SIMD Paradigm

The Single Instruction Multiple Data (SIMD) paradigm is a coarse-grained synchronous form of parallelism. That means that parallel processes all perform the same computations in lock step, each on different data items. In our banking analogy of the cash withdrawal example we now have each desk performing all of the subtasks, but completely synchronized. You clearly see the difference with the vector paradigm. The amount of work for each process is much larger. Again, without load imbalance the speedup can have its ideal value. Prove this yourself. Also, try to find expressions for the speedup when load imbalance is present, so in the case that $m \neq n$.

The SIMD paradigm was very popular some 10 years ago, because the largest parallel computers at that time had SIMD hardware. This paradigm also works quite well for regular problems. Today, SIMD is considered a bit too restrictive to express parallelism in general applications.

2.3.5 The MIMD Paradigm

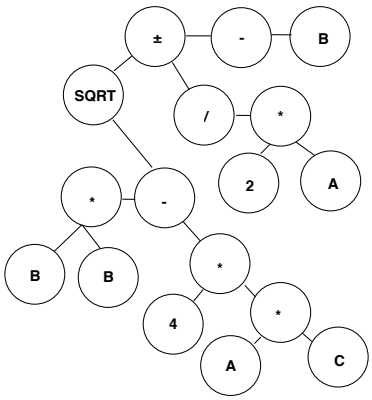
Let us now move our attention to asynchronous parallelism, and first concentrate on the coarse-grained Multiple Instruction Multiple Data (MIMD) parallelism. Here, parallel processes execute completely asynchronous. This means that in our bank, each desk is operating independent from all other desks, on a multitude of tasks. There is no central

clock, and all desks can perform different actions. However, this is not the same as the farming parallelism from Section 2.3.2. We can no longer assume that all clients are completely independent, or that all actions from all desks are independent. For instance, it might happen that two desks want to access the same account, or, two processors want to access the same data item in memory. This may lead to inconsistencies and unacceptable errors. Therefore, consistency and data integrity must be guaranteed by providing synchronization mechanisms, e.g. mutual exclusion through locks on data items in a shared memory, or copies of data items available at each processor and keeping this data consistent by exchanging messages.

The MIMD paradigm is the most general form of parallelism, and can be used to allow multiple heterogeneous tasks to be executed at the same time on a general heterogeneous HPC platform. As we will see, the MIMD paradigm is too general, especially for massive parallelism. Another paradigm, the so-called Single Program Multiple Data paradigm, is more suited. However, before moving to this very important paradigm, we first finalize our discussion, and move to a fine-grained form of asynchronous parallelism, the dataflow paradigm.

2.3.6 The Dataflow Paradigm

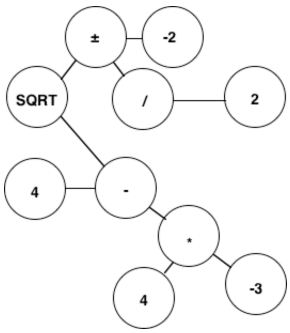
The dataflow paradigm is a fine-grained form of asynchronous parallelism. Here, fine-grained processes are carried out if data is available or if data is needed. A computation is represented as a directed graph, and computations are carried out as soon as data is available, or as soon as data is required. Hence, the dataflow paradigm is based on a graph reduction model which is driven by the availability of data or by the demand for data. In the first case we speak of *eager evaluation* in a true dataflow paradigm, in the second case we speak of *lazy evaluation* or the *reduction* paradigm.

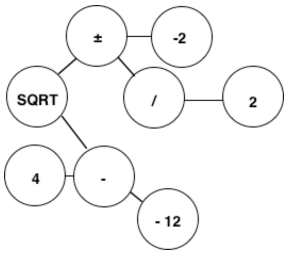


We only show a simple example of the reduction paradigm. Suppose we must calculate the roots of a second order polynomial, which are

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

This computation is represented by the graph on the left. In the reduction paradigm a task is enabled if its results are required by another task which was already enabled. Enabled tasks execute if all their input data is available. These tasks can be executed asynchronously in parallel. The data integrity is guaranteed by the process of demand driven graph reduction. Now suppose that $a = 1$, $b = 2$, $c = 3$, and somewhere the roots of the polynomial are needed, i.e. the result of the / operation is the graph is requested. This means that all tasks in the graph get enabled and are reduced. Below we show a sequence of how these reductions are carried out. Try to follow the process in detail.





Although this paradigm holds an exiting promise for automatic extraction of parallelism, it has actually never left the laboratory stage. The overheads that are included in actually building reduction or dataflow machine are just to high. Nevertheless, many efforts are still being put into this fine-grained form of parallelism.

2.3.7 The SPMD Paradigm

Looking at coarse-grained parallelism, the MIMD paradigm was a bit too general, whereas the SIMD paradigm was a bit too tight. Over the years a new paradigm emerged, that we now know as the Single Program Multiple Data (SPMD) paradigm. This paradigm is most widely used for developing massively parallel programs, and in the rest of this course we will also concentrate on SPMD parallelism.

The idea is to have one single program that operates asynchronously on different data items. Depending on the control in the program (e.g. if-then-else constructs or stop criteria for loops) each instance of the program may go through a different execution path. Synchronization of the parallel parts is established by e.g. an iteration counter, locks on shared data items, or exchange of messages. Currently several programming environments to write SPMD programs exist. The most important one, which will be introduced later, is the Message Passing Interface (MPI).

The SPMD paradigm, which naturally maps onto MIMD architectures (why?) is most widely used within the HPC community. This is also the paradigm that will be discussed in more detail in the rest of the course. However, currently we see an increasing popularity of shared memory systems, which allow another paradigm, so-called data parallelism. For more information on this topic we refer to e.g. Ref. [31].

2.4 Conditions for Parallelism

2.4.1 Dependencies

Now that we know which forms or parallelism exist we should ask the question *when* processes can be executed in parallel. As with paradigms for parallelism, you will probably again have an intuitive idea about this question. Loosely speaking, processes can execute in parallel when they are independent of each other. We will investigate three forms of dependencies:

- Data dependencies;
- Control dependencies;
- Resource dependencies.

If processes do not dependent on each other in any of these three ways they can be executed in parallel.

First look at data dependencies. Each process has data as input and data as output. Looking at the input and output of two processes, S_1 and S_2, the following forms of data dependence are distinguished.

1. *Flow dependence*: S_2 is flow dependent on S_1 if an execution path exists from S_1 to S_2 and if at least one output of S_1 feeds as input to S_2. Flow dependence is denoted as $S_1 \longrightarrow S_2$. As an example, suppose S_1: x := a, and S_2: y := x. The output variable 'x' of S_1 is an input variable for S_2.

2. *Anti-dependence*: S_2 is anti-dependent on S_1 if an execution path exists from S_1 to S_2 and if at least one output of S_2 is input for S_1. Anti-dependence is denoted as

$S_1 \dashv\!\rightarrow S_2$. As an example, suppose S_1: y := x, and S_2: x := a. The output variable 'x' of S_2 is an input variable for S_1.

3. *Output dependence*: S_1 and S_2 are output dependent if they produce the same output variable. Output dependence is denoted as $S_1 \dashv\circ\rightarrow S_2$. As an example, suppose S_1: y := a, and S_2: y := b. The output variable 'y' of S_1 is also an output variable of S_2.

4. *I/O dependence*: As a special case of output dependence, if the same file is referenced by two processes, they are I/O dependent.

As an example, consider the following program, consisting of 4 processes:

S_1 : `Load` R_1, A i.e. copy the content of memory location A to register R_1.

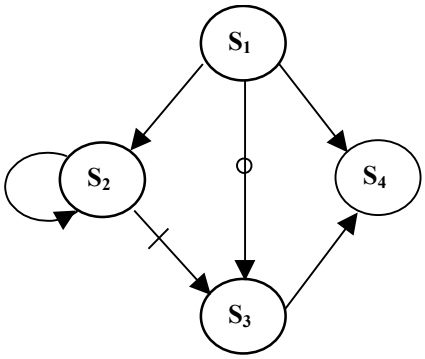
S_2 : `Add` R_2, R_1 i.e. add the contents of registers R_1 and R_2 and put the result in register R_2.

S_3 : `Move` R_1, R_3 i.e. move the content of register R_3 to register R_1.

S_4 : `Store` B, R_1 i.e. copy the content of register R_1 to memory location B.

We will find all data dependencies among these four processes and next draw a dependency graph. The easiest way to do this is to organize everything into a dependency matrix and go through all pairs of processes one by one, like in the matrix below.

	S_1	**S_2**	**S_3**	**S_4**
S_1	no dependencies	$\longrightarrow \{R_1\}$	$\dashv\circ\rightarrow \{R_1\}$	$\longrightarrow \{R_1\}$
S_2		$\longrightarrow \{R_2\}$	$\dashv\!\rightarrow \{R_1\}$	no dependencies
S_3			no dependencies	$\longrightarrow \{R_1\}$
S_4				no dependencies



On each row of the matrix we analyze the data dependencies of a process with all other processes that are executed after the process in the program. So, for e.g. S_2 we only check the dependencies with S_3 and S_4. Once a data dependency is found, the corresponding symbol, is put in the matrix, together with the variable that causes the dependency. Note that more then one data dependency between processes is possible. Next we draw the data dependency graph, shown here on the left. As a convention, we also look at data dependencies among the input and output of one single process.

Another type of dependency can be introduced through the control flow of a program. In many cases one cannot predict the exact execution order in advance, due to the control flow of the program. Consider the following two loops:

```
For (i = 0; i <= N; ++i ) {
   A[i] = C[i];
   If (A[i] <= 0)
         A[i] = 1
}
```

and

```
For (i = 0; i <= N; ++i ) {
   If (A[i-1] <= A[i])
         A[i] = 1
}
```

The first loop is control independent, because each iteration of the loop does not depend on other iterations. The second loop however *is* control dependent because the iterations depend on previous iterations, and therefore we cannot execute all iterations in parallel.

Finally, two processes are resource dependent if an execution path between both processes exists and if they use the same resources. For instance, the processes S_1 : y = a + b and S_2 : z = c + d both use an adder unit, and are therefore resource dependent.

2.4.2 Bernstein Conditions

Bernstein conditions express when two processes can be executed in parallel, and are a mathematical formulation of the phrase that processes should not have data dependencies. Let I_i be the input set of a process S_i containing all input variables needed to execute S_i, and likewise let O_i be the output set of S_i. Two processes S_1 and S_2 can be executed in parallel, denoted by $S_1 \parallel S_2$ if

$$I_2 \cap O_1 = \emptyset, \quad [7.a]$$

$$I_1 \cap O_2 = \emptyset, \quad [7.b]$$

$$O_1 \cap O_2 = \emptyset. \quad [7.c]$$

The first Bernstein condition (Eq. 7.a) expresses that both processes should not have a flow dependence, the second condition (Eq. 7.b) forbids anti-dependence, and the third (Eq. 7.c) forbids output dependence. Suppose we have n processes S_i, with $1 \leq i \leq n$, and the sequential executing order is from 1 to n. These processes can run in parallel if and only if $S_i \parallel S_j$ for all $i < j$. This results in a total of $3n(n\text{-}1)/2$ conditions that should be satisfied.

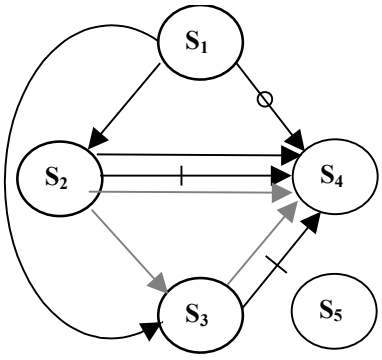
Consider the following five processes:

S_1 : C = D x E
S_2 : M = G + C
S_3 : A = B + C
S_4 : C = L + M
S_5 : F = G / E

The input and output sets of these processes are:

I_1 = {D,E} O_1 = {C};
I_2 = {G,C} O_2 = {M};
I_3 = {B,C} O_3 = {A};
I_4 = {L,M} O_4 = {C};
I_5 = {E,G} O_5 = {F}.

Next, we create a dependency matrix, and go through all Bernstein conditions. The resulting matrix and data dependency graph are shown below (as an exercise, try to derive them yourself).



From the matrix we can conclude that $S_1 \parallel S_5$, $S_2 \parallel S_3$, $S_2 \parallel S_5$, $S_3 \parallel S_5$, and $S_4 \parallel S_5$. Other combinations are not possible due to data dependencies. For instance, between S_1 and S_2 there is a flow dependence due to the variable C. However, we should also look at possible resource dependencies. Assuming that we have separate hardware units for addition, multiplication, and division, we immediately detect resource dependencies between S_2 - S_3, S_2 - S_4, and S_3 - S_4 due to the addition. The resulting dependency graph is shown on the left, where the gray lines denote the resource dependencies.

	S_1	S_2	S_3	S_4	S_5
S_1	$I_1 \cap O_1 = \emptyset$	$I_2 \cap O_1 = \{C\}$ $I_1 \cap O_2 = \emptyset$ $O_1 \cap O_2 = \emptyset$	$I_3 \cap O_1 = \{C\}$ $I_1 \cap O_3 = \emptyset$ $O_1 \cap O_3 = \emptyset$	$I_4 \cap O_1 = \emptyset$ $I_1 \cap O_4 = \emptyset$ $O_1 \cap O_4 = \{C\}$	$I_5 \cap O_1 = \emptyset$ $I_1 \cap O_5 = \emptyset$ $O_1 \cap O_5 = \emptyset$
S_2		$I_2 \cap O_2 = \emptyset$	$I_3 \cap O_2 = \emptyset$ $I_2 \cap O_3 = \emptyset$ $O_2 \cap O_3 = \emptyset$	$I_4 \cap O_2 = \{M\}$ $I_2 \cap O_4 = \{C\}$ $O_2 \cap O_4 = \emptyset$	$I_5 \cap O_2 = \emptyset$ $I_2 \cap O_5 = \emptyset$ $O_2 \cap O_5 = \emptyset$
S_3			$I_3 \cap O_3 = \emptyset$	$I_4 \cap O_3 = \emptyset$ $I_3 \cap O_4 = \{C\}$ $O_3 \cap O_4 = \emptyset$	$I_5 \cap O_3 = \emptyset$ $I_3 \cap O_5 = \emptyset$ $O_3 \cap O_5 = \emptyset$
S_4				$I_4 \cap O_4 = \emptyset$	$I_5 \cap O_4 = \emptyset$ $I_4 \cap O_5 = \emptyset$ $O_4 \cap O_5 = \emptyset$
S_5					$I_5 \cap O_5 = \emptyset$

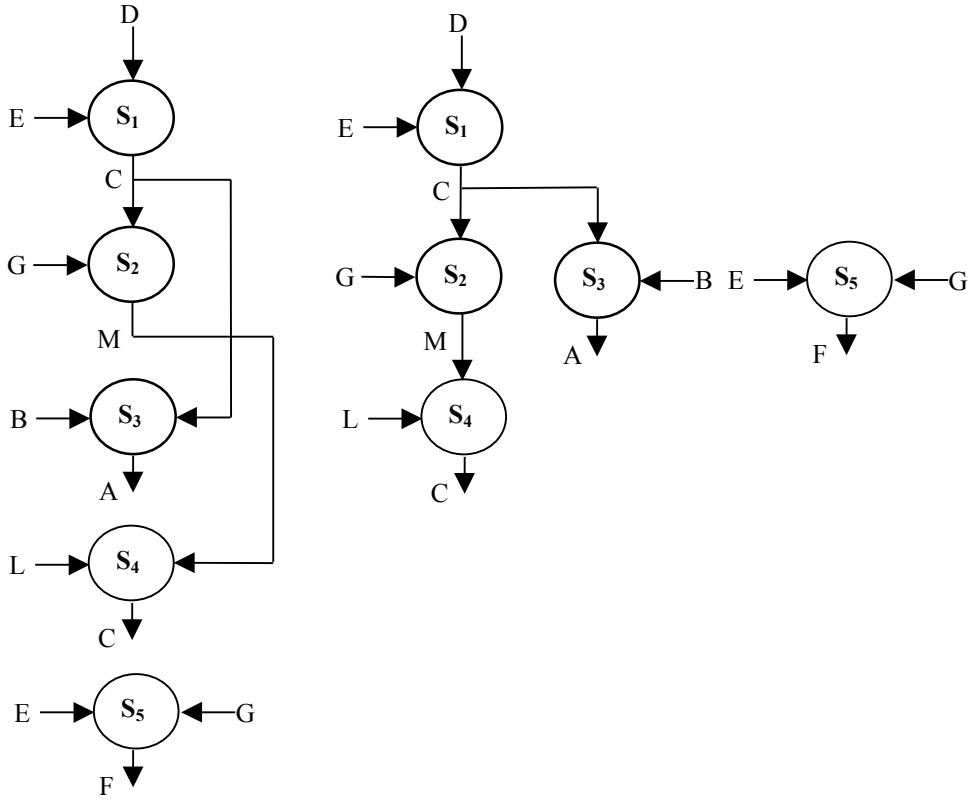


Figure 13 : Left, the original sequential code, and right, the parallel version.

With all information on the dependencies we can try to reconstruct the computation such that the parallelism is exploited as much as possible. Assume that two adder units are available. In that case S_2 can be executed parallel with S_3. Figure 13 shows the original sequential order of the 5 processes, together with the input and output variables, and also the restructured parallel code. If each instruction takes a time τ, the sequential execution time is 5τ, the parallel execution time 3τ, and the speedup is 5/3.

2.5 Parallelization

2.5.1 Complex Systems

We are now ready to ask the question *how* a large sequential computation can be transformed in a parallel computation. We will show that parallelization of a computation is accomplished by *decomposition*. Before going into that subject, we will first introduce the concept of modeling and simulation as a mapping of *complex systems*. In that context parallelization is nothing but a mapping of one complex system (a model) to another complex system (a parallel computer).

First, let us briefly recall the steps that are taken in going from a natural phenomenon to a simulation on a computer (see Figure 14). Suppose we try to predict the motion of planets in our solar system. So, our natural phenomenon is the sun, the nine planets with their moons, and all the asteroids, comets, and dust that are floating around. Furthermore, the solar system is immersed in the Milky Way, which again is immersed in the cosmos.

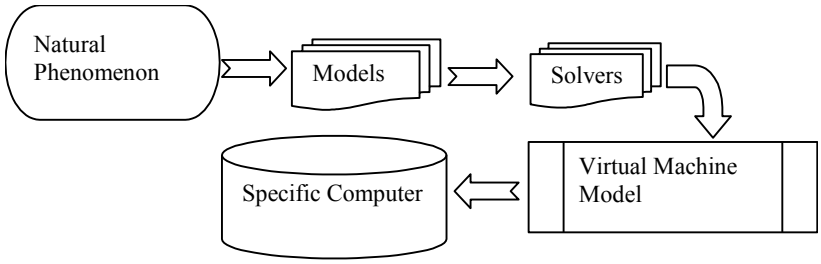


Figure 14 : Modeling and simulation of natural phenomena

The first step is to define a model that is suitable to answer our question, i.e. a prediction of the motion of the planets. Let us assume that the influence of other stars in the Milky Way and other galaxies in the cosmos can be completely ignored. Furthermore, we also assume that the influence of the asteroids, comets, and dust in the solar system on the motion of the planets can also be ignored. Finally, we ignore the moons that circulate the planets. Of course, such assumptions should be verified, that is, within the wanted accuracy of the model one should find arguments that the model assumptions are allowed. We now have reduced our natural phenomenon to a model consisting of 10 massive bodies that attract each other through gravity. As the distances between the planets and between the planets and the sun are much larger than their diameters, we also assume that the sun and the planets are point masses. We completely neglect the rotation of the sun and the planets around their own axes as well. Finally, we assume that we can neglect relativistic effects. Our model is now reduced to 10 point masses attracting each other through Newtonian gravity. The motion of the point masses is determined by Newton's second law, i.e. $\mathbf{F} = m\mathbf{a}$, where $\mathbf{F}$ is the total gravitational force on a planet, m is its mass and $\mathbf{a}$ is the resulting acceleration.

The next step is to try to solve the model. If the model would contain two bodies (e.g. the Sun and the earth) it can be solved analytically. However, for models containing three or more bodies, no analytical solutions are known. If we want to proceed we should either simplify our model or turn to explicit simulation on a computer.

To simulate our model of the solar system, we first need to transform it into a form that is suitable for execution on a computer. This transformation results in the solver layer in Figure 14. Let us take a closer look at Newton's second law. In our model we should write it as

$$m_i \frac{\partial^2 \mathbf{x}_i}{\partial t^2} = \sum_{\substack{j=1 \\ j \neq i}}^{10} \mathbf{F}_{ij} , \qquad 1 \le i \le 10 , \qquad [8]$$

where the subscripts i and j denote the sun or one of the planets, m is the mass, $\mathbf{x}$ the position, t the time, and $\mathbf{F}_{ij}$ the gravitational force between body i and j, which is calculated as

$$\mathbf{F}_{ij} = -Gm_i m_j \frac{(\mathbf{x}_i - \mathbf{x}_j)}{\left|\mathbf{x}_i - \mathbf{x}_j\right|^3}, \qquad [9]$$

with G the gravitational constant. If all masses are known, and the initial positions and velocities of the bodies at time $t = 0$ are specified, we can find the orbits $\mathbf{x}_i(t)$ of the bodies by integrating Eq. [8] in time. The differential equation [8] must be discretized such that we can integrate it numerically. The time axis is divided in intervals Δt, and we seek solutions of Eq. [8] on times $t_k = k\Delta t$, with $k = 0, 1, 2 \cdots$. In section 3 we show how such a discretization can be done. For now it is sufficient to know that in order to calculate $\mathbf{x}(t_k + \Delta t)$ one must first calculate the forces on time t_k, using Eq. [9], and next advance all positions from $\mathbf{x}(t_k)$ to $\mathbf{x}(t_k + \Delta t)$. In order to have an accurate simulation of the orbit one should choose a small time step Δt.

Next, we transform our solver to a virtual machine model, i.e. to a computer program in which the positions of the bodies are represented by abstract data structures, and the discrete time stepping formulas are coded as an algorithm using single or double precision arithmetic.

Finally, the source code is compiled and linked for a specific computer. The data structures are assigned to memory locations and the algorithm is executed, hopefully in an optimized way, by a processor.

It should now be clear that each step in Figure 14 involves an abstraction and a mapping of one domain to another. Let us now move away from our specific example and define a more general framework.

Define a Dynamic Complex System (DCS) as a (large) set of members with, in general, dynamic connections between them. A DCS evolves according to deterministic of statistical rules. The interconnections can be geometrical, generated by forces, biological channels, symbolic, social interaction, etc. It is easy to come up with examples of DCS's. For instance, a lecture room with students and a teacher is a DCS. Traffic on the road, in free flow or stuck in a traffic jam, is a DCS. A cellular automaton, such as Conway's game of life, is a DCS. The stock market is a DCS.

Dynamic Complex Systems

Research to Dynamic Complex Systems is a hot topic. Especially the dynamical behavior of a DCS gets much attention. The exiting thing here is so-called emergent behavior. The entities and their connections can be very simple, but their collective dynamic behavior can be very complex and completely unexpected if you only look at the simplicity of the entities. Conway's game of life serves as a good example. The rules of this cellular automaton are extremely simple. However, if you let it evolve, you can observe all kinds of strange behavior, like the `gliders' or the `blinkers'. Research towards this emergent behavior of DCS is a very exiting field.

Interestingly, if we look at our example of simulation of the solar system, each layer in Figure 14 can be considered as a DCS. For instance, in the final model the entities are the 10 bodies and the interconnections the gravitational forces. In the virtual machine model layer, the entities are the abstract data structures and the interconnections are the arithmetic operations. Formally, all steps in Figure 14 may be viewed as a mapping between DCS's. For instance, if we call the DCS in the model layer DCS_m and in the solver layer DCS_s, we could write

$$DCS_S = \Gamma_{m \to s}(DCS_m), \qquad [10]$$

where $\Gamma_{m \to s}$ is a formal mapping function that connects the model layer with the solver layer. An interesting question of course would be if we can actually find such mapping functions and apply them to large classes of DCS.

We did not yet mention one important DCS, a parallel computer! Clearly, a parallel computer is a DCS, where the processors are the entities and the interconnections are formed by the interconnection network or shared memory. So, if we want to simulate our model on a parallel computer, we should find a mapping between the model DCS and the parallel computer DCS. In the next section we will show that in order to do this we need to include one extra layer in Figure 14, between the solver and virtual machine model layer: the decomposition layer.

2.5.2 Decomposition

We take the viewpoint that parallelization is a mapping from one DCS representing a model or solver, to another DCS representing a parallel virtual machine model. This allows us to loosely describe the act of parallelization as splitting a model into parts, by taking groups of its entities together, and assigning such groups to processors. This is called decomposition. Let us see how this works out in the example of the solar system. In order to keep drawings manageable we reduce the model to only 4 bodies,. Furthermore, assume that we have a distributed memory computer with only two processors connected to each other via a network. The DCS representations of the model and the parallel computer are drawn in Figure 15.

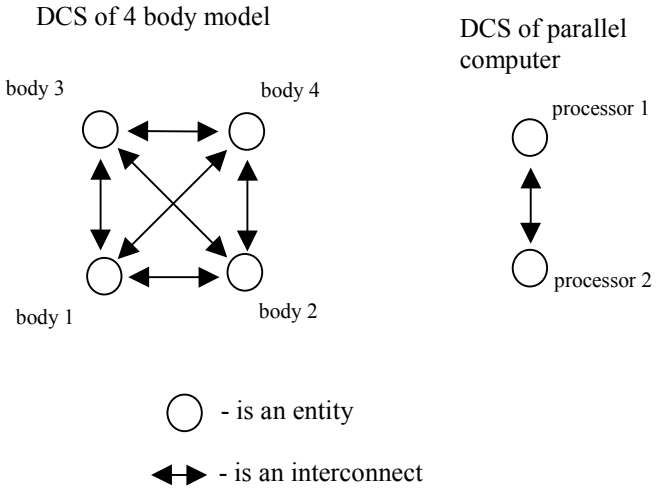


Figure 15 : DCS representation of the 4 body model and the 2 processor parallel computer.

We now first decompose the problem, by e.g. taking body 1 and 2 together, and body 3 and 4 together. This is drawn in Figure 16. The resulting DCS representing the decomposed problem is trivially mapped to the parallel computer, because the DCS representation of the decomposed problem and of the parallel computer is equal. Grain 1 is mapped to processor 1 and grain 2 to processor 2. The entities in the DCS representation of the decomposed model contain many entities of the original DCS representation of the model. We call these entities grains, and this is another way of looking at the grain size of a parallel computation. You already encountered grain size in section 2.3. Coarse - and fine-grained parallelism is now related to the amount of model entities per grain.

In our flow chart of modeling and simulation we need to include the decomposition layer. This results in Figure 17. In our model, the mapping from the decomposition DCS to the virtual machine model DCS was trivial, because their representations were equal. However, in many cases this will not be the case, and this mapping will be very complicated matter. In fact, it is known that in general this formal mapping is NP-

complete and that one needs highly specialized techniques to solve this problem. Fortunately, in many applications mapping is easily accomplished.

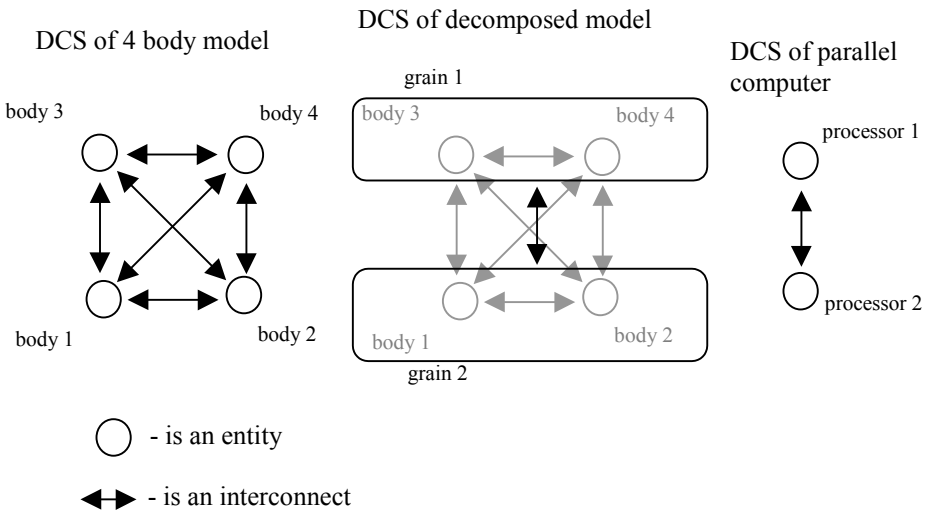


Figure 16: Decomposition of the 4 body model, and resulting DCS of the decomposed model. In the DCS representation of the decomposition, the DCS representation of the model is drawn in gray, to clarify the decomposition.

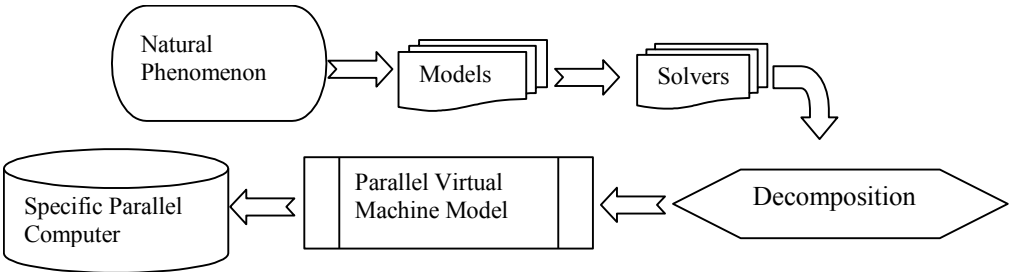


Figure 17: Modeling and simulation, including the decomposition layer.

2.5.3 Types of Decomposition

We distinguish two main types of decomposition, functional - and data decomposition. In functional decomposition the grains in the DCS representation of the decomposed model are operations (functions) and the interconnections are formed by the data that go through the functions. On the other hand, in data decomposition the grains are chunks of data, and the interconnections are due to operations on data items. You could go back to section 2.3 and apply the above definitions to the example of the bank. Below we introduce another analogy to clarify the concepts of functional - and data decomposition.

Suppose you want to build a wall of a certain length, and you are allowed to hire as many bricklayers (i.e. processors) as you want. The data items are the bricks and the functions on the data items are preparing mortar, applying the mortar to the bricks, and finally laying the bricks to form the wall.

First look at functional decomposition. The entities are the three functions, and the interconnections are the bricks moving through the cycle. We immediately recognize the vector paradigm of parallelism in this example. Other types of functional decomposition are also possible, where there is not such a strong connection between the different functions. However, as a general rule, the amount of parallelism offered by functional decomposition is usually not very large (in this example three). Therefore, to obtain massive parallelism, the other alternative is usually applied. We will return to this point later in this section.

Now turn to data decomposition. This also comes under the name of domain decomposition or geometric decomposition. The grains in the DCS representation are formed by chunks of data, and the interconnections are due to operations on that data. In wall building language, we split up the wall (the domain, the data volume) in pieces, and each bricklayer is assigned to one of the pieces of the wall. Figure 18 shows the data domain. We should realize that several forms of data decomposition are possible.

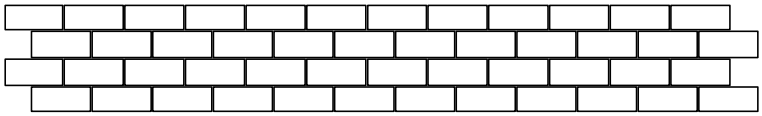


Figure 18: A piece of the data domain, i.e. a piece of the wall.

First, one could decide to do layered decomposition, as in Figure 19. This is not a very good choice, because the number of layers now limits the amount of parallelism. That means that the scalability of this decomposition is very bad. In case that you have more bricklayers than layers in the wall (i.e. more processors than layers) the extra bricklayers cannot do any work and remain idle. In other words, this form of data decomposition is not scalable.

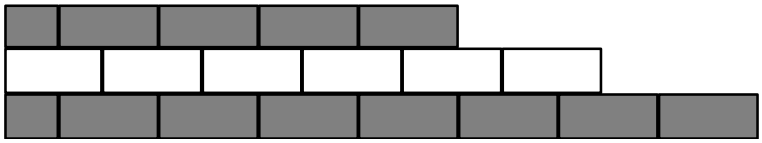


Figure 19: A layered data decomposition.

A better way is to equally divide the length of the wall over the available number of bricklayers. This results in the data decomposition of Figure 20. This looks much better. The amount of parallelism is not really limited (well, yes, by the length of the wall, but assume that it is really long). The grains are now the pieces of the wall. Notice that if we assume that each brick needs the same amount of work and that all bricklayers are equally fast, that with the current domain decomposition we have taken care of a good load balancing. Generally speaking, good data decomposition should result in a scalable parallelism with good load balancing.

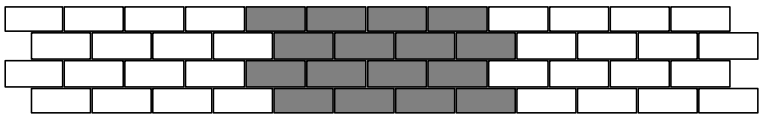


Figure 20: A vertical data decomposition of the wall.

What about the interconnections between the grains? In this example they are caused by what happens at the boundaries of the domains. In order to lay the bricks that make up the boundary of the domain, communication with the neighboring domain is needed. If, as in Figure 21, brick B should be applied, this is only allowed if brick A is already present. This means that there is communication between both domains, and that laying bricks in the boundary of a domain depends on the work in the neighboring domain. Note that if the grains are made larger, the amount of fully internal points, where no communication with neighboring processors is needed, grows whereas the number of boundary points stays constant. This means that in this case large grain sizes reduce the communication overheads and may result in better speedups (but not necessarily in a faster time needed for building the complete wall, think about this point, it is very important!).

Note that the wall building example is not just a toy problem. A large class of very relevant models results in comparable parallelism as our wall, with domains with independent internal points and some minor communication on the boundaries of the domains. In section 3 you will encounter an example of such an application.

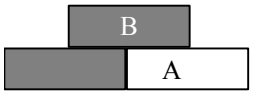


Figure 21 : Interconnections between two domains.

So far data decomposition has been easy, because we assumed that the wall is completely regular. That made the data decomposition easy. Each bricklayer got an equal part of the full length of the wall, thus ensuring good load balancing. Now assume an irregular domain, as in Figure 22. The domain decomposition is no longer trivial. In splitting up the wall one should define some algorithm to guarantee load balancing. Try to define such an algorithm yourself.

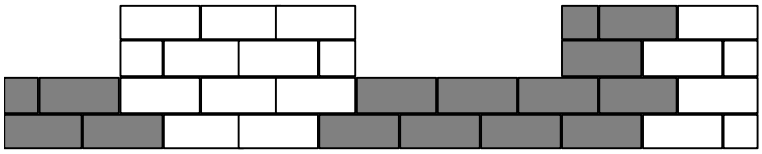


Figure 22: A wall with an irregular domain.

Next, to make things even more complicated, we assumed that all bricks where equal, and needed the same amount of processing time. In general this is also not the case. It might well be that some bricks need other processing time than others. In that case load balancing becomes even more difficult. It might even be that the amount of processing is not known in advance, or that during the execution the processing time changes dynamically for each entity. In this case it becomes necessary to change the decomposition *during* the execution of the parallel program, leading to the very complicated matter of *dynamic load balancing*. Finally, to show you the full complexity, we also assumed that all processors are equally fast. In a heterogeneous systems this is not the case. Furthermore, the speed of the processors can also change dynamically, due to unpredictable load induced by other users (this happens e.g. if you would run your parallel program on a network of workstations). Trying to keep the parallel execution efficient, in such a way that the resources are used in an optimal way, and that the turn around time of applications is minimized, is still part of active research within the HPC community. These issues are part of more advanced lectures (“Parallel Wetenschappelijk Rekenen & Simulatie).

Data decomposition leads to massive parallelism, and is usually implemented in the outer loops of a program. Functional decomposition usually has a limited form of parallelism that could however be exploited in inner loops. Below we show a piece of pseudo code for our parallel wall building, where both types of decomposition are used. Typically, a compiler can vectorize the inner loop, and the outer global parallelism is introduced by explicit domain decomposition.

```
proc_begin main (Construct the wall)
   for_begin (Each section of the wall) [Global Parallelism]
      for_begin (Each row of bricks)
         for_begin (Each brick in row) [Local Parallelism]
            [Prepare mortar]
            [Apply mortar]
            [Add brick to wall]
         for_end
      for_end
   for_end
proc_end
```

2.5.4 A First Look at Performance

How should we characterize the performance of a parallel computation? Of course, the bottom line is the wall clock time that is needed to complete the job, as in sequential computations. Usually we seek to minimize the wall clock time, or we try to solve the largest possible problem in a fixed amount of time. However, only wall clock time is not sufficient. We want to define metrics that allow us to analyze in detail the quality of parallelization, or the sources that result in reductions of the computational speed. The issue of performance analysis is very important, and in section 4 it is treated in much detail. Here, we just introduce some first basic issues.

Define T_1 as the execution time of a parallel algorithm on one processor, T_p as the execution time of the parallel algorithm on p processors, and T_{seq} as the execution time of the most efficient sequential algorithm solving the same problem on one processor. Note that $T_{seq} \leq T_1$. This must be the case, because if we find $T_1 < T_{seq}$ then the parallel program on one processor would automatically become the most efficient sequential program. We define the *relative speedup* as

$$S_p = \frac{T_1}{T_p}, \qquad [11]$$

the *fair speedup* as

$$\bar{S}_p = \frac{T_{seq}}{T_p}, \qquad [12]$$

the *relative efficiency* as

$$\varepsilon_p = \frac{S_p}{p} = \frac{T_1}{pT_p}, \qquad [13]$$

and the *fair efficiency* as

$$\bar{\varepsilon}_p = \frac{\bar{S}_p}{p} = \frac{T_{seq}}{pT_p}. \qquad [14]$$

The relative speedup was already encountered in previous sections, where we just called it speedup. This is what usually happens and we will also do this. However, it is good practice to make a clear distinction between relative - and fair speedup. In many cases several algorithms exist to solve to same problem, and the most efficient sequential algorithm is very difficult to parallelize. Another, less efficient algorithm might be easy to parallelize. Therefore, one could decide to use the less efficient sequential algorithm as the basis for parallelization. However, in trying to obtain the benefits of parallelization, one should of course compare the execution times with the most efficient sequential algorithm, i.e. measure fair speedup and fair efficiency. On the other hand, in many cases you don't know the most efficient sequential algorithm, or you just do not have execution times available, and you are forced to measure the relative speedups.

Relative speedup and relative efficiency are important internal performance metrics, in the sense that they express how well an algorithm is parallelized and how well it scales to a large amount of processors. Relative speedup and efficiency are bounded by

$$1 \leq S_p \leq p,$$

$$\frac{1}{p} \leq \varepsilon_p \leq 1.$$

Try to prove these bounds yourself. A good parallel program has efficiency close to one. The parallel program is said to be scalable if it maintains a high efficiency while the number of processors is increased. In chapter 4 we will further discuss the issue of scalability.

Usually the efficiency will be smaller than one, due to all kinds of overheads. In fact, efficiency measures the amount of cycles that we loose due to overheads that are introduced because the program runs in parallel. We already encountered some of those overheads, such as load imbalance and the necessity to communicate between processors. It is useful to express the overheads relative to T_1, leading to the concept of fractional overheads. As an example, assume that we have parallelized a sequential program (the fastest known) such that the only overheads are due to communication between processors, and that each processor spends a time T_{comm} for communication. We write

$$T_1 = T_{seq} ,$$
$$T_p = \frac{T_1}{p} + T_{comm} .$$

Using Eq. [13] we find

$$\varepsilon_p = \frac{1}{1+f_c} , \qquad [15]$$

where f_c is the fractional communication overhead,

$$f_c = \frac{pT_{com}}{T_1} . \qquad [16]$$

For other overheads comparable equations can be found. If the overheads are small, we may Taylor expand Eq. [15] around $f_c = 0$, and to first order we find

$$\varepsilon_p = 1 - f_c . \qquad [17]$$

As an example, consider a parallel vector addition, where two vectors of length n are added and the results stored in a third vector. Assume that one processor, say the processor with identification number 0, reads the two vectors from disk. Each processor receives from processor 0 a part of the vectors, the parallel addition is performed, and all processors send the result back to processor 0. Assume that an addition takes a time τ_{calc} and that sending one element of the vector takes a time τ_{comm}. Furthermore, assume that processor 0 can only communicate with one other processor at the time. With these definitions and assumptions we find

$$T_1 = n\tau_{calc} , \qquad [18.a]$$

$$T_{comm} = 3(p-1)\frac{n}{p}\tau_{comm} , \qquad [18.b]$$

$$f_c = 3(p-1)\frac{\tau_{comm}}{\tau_{calc}} . \qquad [18.c]$$

This form of f_c is typical. It has in it the quotient τ_{comm}/τ_{calc} which are hardware parameters of the specific parallel computer. Furthermore, the amount of processors appears. If p becomes larger, so does f_c and therefore increasing p leads to a decrease of ε_p. Finally, f_c contains an element of the parallel algorithm. The number 3 is because of the specific communication structure in the parallel algorithm.

Finally, look what happens with the execution time T_p if we increase p in our parallel vector addition. The communication time (Eq. 18.b) becomes constant (in the limit of

infinite p) whereas the calculation part (T_1/p) goes to zero. This means that adding more processors does not lead to any faster execution. We say that this algorithm is communication bounded for large a number of processors, because most of the execution time is spent sending data from processor 0 to all other processors. We can in fact improve this situation, either by changing the communication structure and/or by lifting a constraint of the parallel hardware. Try to do this yourself and try to reduce f_c as much as possible.

2.6 Message Passing

2.6.1 Introduction

Section 2.3.7 introduced the SPMD paradigm. For massively parallel computing this paradigm is most widely used. Furthermore, in section 2.2.1 we introduced the distributed memory parallel architecture. Again, for massively parallel processing this is the typical choice of hardware. In the sequel we will concentrate on SPMD parallelism executed on distributed memory parallel computers. This automatically leads to the necessity of *Message Passing* between processors. Before moving to this very important topic, we first want to remind you that besides SPMD parallelism another important paradigm exists, that of Data Parallelism. This type of parallelism can be supported very efficiently on shared memory computers. However, currently it is not efficiently supported on distributed memory computers. On the other hand, message passing SPMD programs can in fact be executed very efficiently on shared memory computers.

SPMD parallelism, and also the more general MIMD parallelism, executed on distributed memory computers, gives rise to message passing, i.e. the exchange of messages between processors. Consider for instance farming parallelism, where a master processor sends work items to slave processors, see Figure 23. A slave processor finalized a piece of work, and sends a message to the master, announcing that it is out of work. The master processor in its turn sends a message back to the slave, containing a new piece of work.

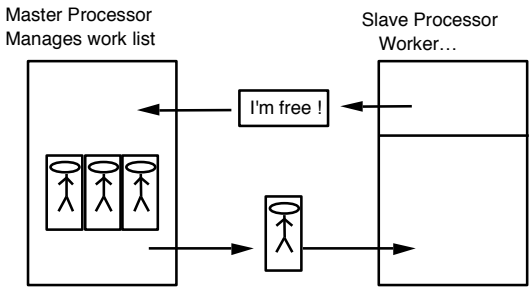


Figure 23: Message Passing in a farming application.

Physically, a message is a stream of bytes taken from one processor's memory, passed through some communication medium, and placed in another processor's memory. This lecture will not discuss the communication hardware, nor low level software that is needed to efficiently implement certain types of message passing. We will consider, from a conceptual viewpoint, the main types of message passing.

2.6.2 Message Passing Primitives

In message passing between two processors we must always distinguish between the processor that sends the data, i.e. the Sender, and the processor that receives the data, i.e. the Receiver. We distinguish four main types of message passing primitives. Message passing can *synchronous* or *asynchronous* and *blocking* or *non-blocking*. For the Sender, all four types are relevant, for the Receiver only the distinction between blocking and non-blocking message passing is relevant, see Table 5.

Sender	Receiver
Blocked Synchronous	Blocked
Blocked Asynchronous	
Non-blocked Synchronous	Non-blocked
Non-blocked Asynchronous	

Table 5: Main types of message passing primitives.

The use of blocking or non-blocking communication affects the way in which the transmission function determines that the communication has *completed*. By using either synchronous or asynchronous communication, we specify what is meant by *completion*. When a message is transmitted synchronously, the transmission is completed only when the receiving process has received the message in its own local buffer. When a message is transmitted asynchronously, the message transmission is completed when the communication network has received the data. A blocking communication call will always suspend the calling process until the transfer has been completed, whereas a non-blocking communication call will return almost immediately. With a non-blocking transfer we must use an additional (testing) function to determine if the communication has been completed.

Notice that although from a conceptual point of view the definitions are clear, they leave an enormous amount of freedom for those who actually build a message passing system. For instance, from a software point of view, a significant amount of administration has to be included to take care of the correct handling of messages and for efficient memory management (creating and deleting message buffers). Current state-of-the-art message passing environments organize message passing in lightweight processes (threads/daemons) residing on the native processors. In this respect we can view threads and daemons as building blocks for the inter-processor communication network layer. From a hardware point of view, the messages have to be routed through the physical network. Routing hardware is responsible for passing the information through the shortest possible route in the network. The physical network should be invisible to the application programmer, who designs his processor communication in terms of virtual communication, consisting of virtual processes and virtual channels (i.e. logical links), where the actual physical channels are time shared (multiplexed) by all the virtual channels. The message passing layer takes care of the mapping of virtual communication onto physical communication paths. If you are interested such items you could take a look at Refs. [30, 31].

Let us return to the four types of message passing. It should be clear now that for the Receiver there is no difference between synchronous and asynchronous communication, because for a Receiver the communication is always completed when the message is received in a local buffer. A Receiver must ensure that a Sender can transmit messages whenever it wants to, and must be prepared to accept messages on-demand. If this condition is not satisfied the network may be corrupted. For Senders the distinction between synchronous and asynchronous is important. Messages are copied from the Sender's message buffer into the communication network, transmitted to the Receiver, and copied into the local buffers of the Receiver. In synchronous communication the Sender considers the communication completed if the message has arrived in the local buffers of the Receiver. In the asynchronous case, the Sender assumes that the communication is complete if the message is copied into the communication network. In this case the communication protocol must ensure a correct delivery of the message to the Receiver, because in case of errors the Sender will not re-send the message.

The distinction between blocking and non-blocking is relevant for both the Sender and the Receiver. If they instigate a non-blocking communication they cannot assume that the communication has completed until they test the status of the communication and get a `finished' status. The good thing of non-blocking communication is that it allows you to perform other operations (i.e. useful work) while communication is on its way. In this

way you can perform calculations and communications in parallel, and effectively reduce the amount of communication overhead. This technique, that is sometimes called `latency hiding' or `hiding communication behind computation' may lead to drastic improvements of the efficiency of a parallel program. However, the use of non-blocking communication will result in more complex program code, because now you cannot assume that a transmission of data has completed, and this must be checked if data items are needed or if data items are updated. A non-blocking send effectively queues the transmission for completion at a later time (that depends on the Receiver). The data that are to be transmitted will remain in the Sender's message buffer until the transfer has completed. The message buffer should not be altered, or used as the source for a subsequent transfer, until the transfer has completed. Non-blocking receive effectively queues the receive for completion at a time that is dependent on the Sender. Each queued receive should use a unique message buffer to ensure that messages are not corrupted. A buffer's content should not be modified or used until the corresponding data transmission has completed.

Since it is often very insightful to make drawings of a communication process we create space-time diagrams in which time flows from top to bottom and space is on the horizontal axes. Richard Feynman (yes, the same as in section 2.1.1) introduced such diagrams in theoretical physics to visualize interactions between fundamental particles, and today such diagrams are called Feynman diagrams. Therefore, we call our space-time diagrams of communication Feynman diagrams for message passing.

Assume that we have a blocked synchronous send and a blocked receive, and assume that the Receiver first issues the receive command. The corresponding Feynman diagram is shown in Figure 24.

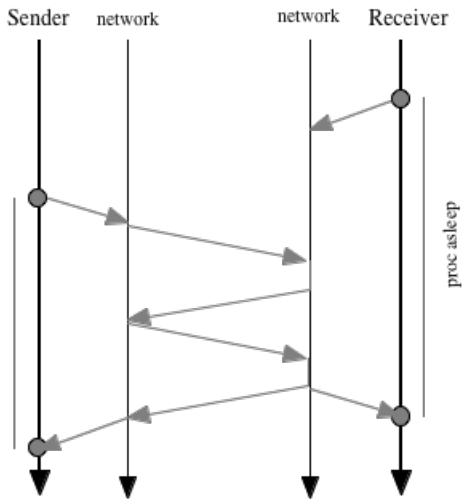


Figure 24: Feynman diagram for blocked synchronous send and blocked receive, with the Receiver first.

The Sender is on the left, the Receiver on the right, and in between the communication network. Arrows denote messages that are sent. Arrows are always pointing down because sending messages takes some time. We assume that the communication network sends the messages in packets of a certain maximum size, and that the message to be sent fits in two packets. Furthermore, we assume that the instigation of the send or receive operation induces some actions to notify the network and setup necessary buffers. This is denoted in the Feynman diagram by the descriptor Q_{desc}. Now, follow the process in Figure 24 in detail. First, the Receiver starts the receive operation, and informs the network (Q_{desc} at the Receiver). Because the receive was blocking, the Receiver process idles, i.e. goes asleep, and waits for a wakeup signal that will be issued after the communication is finalized. Because the Sender did not issue the send command, nothing happens for a while. Next, the Sender issues the send command (Q_{desc} on the Sender). The send operation was also blocking, so the Sender goes asleep and waits for a wakeup call.

Because the send was synchronous, this wakeup call will be issued when the Sender knows that the message is received in a local buffer. The network sends the first packet to the Receiver. After the packet is received and copied into a local buffer, an acknowledgement is returned to the Sender. Now the second packet is sent, copied into the local buffer by the Receiver, and an acknowledgement is returned to the Sender. At the Receiver side, after the second packet is received and copied, the communication is completed and the wakeup call is issued. After the wakeup call, the Receiver process continues its operations. The Sender on the other hand waits until it has received the second acknowledgement, and then gets its wakeup call. Notice that the specific form of the handshake protocol between Sender and Receiver could of course be different in real systems This is just to clarify the main ideas. Furthermore, note that the wakeup of the Sender and Receiver is not necessarily exactly at the same time. A blocked synchronous communication will therefore not lead to exact synchronization of the Sender and Receiver process in the strict sense of the word. Sometimes one says that we have "loosely synchronized" processes and that SPMD programs using blocked synchronous message passing are "loosely synchronous parallel programs".

Next, consider the case of a non-blocked synchronous send and a non-blocked receive, and also assume this time that the Sender process is first. The resulting Feynman diagram is shown in Figure 25.

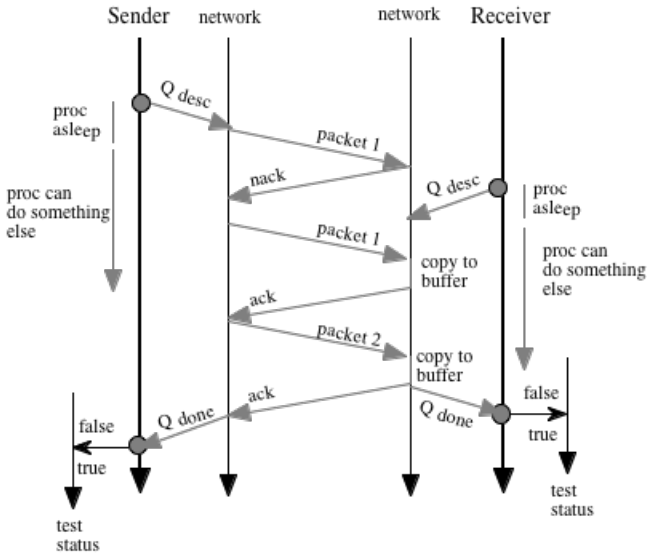


Figure 25: Feynman diagram for non-blocked synchronous send and non-blocked receive, with the Sender first.

First look at the influence of the non-blocking. After the send or receive operation is issued, the network is informed and buffers are prepared (the Q_{desc}), which takes a small amount of time. During this time the issuing process is asleep, but then immediately wakes up again and is free to do something else. The Sender and Receiver can now test the status of the communication, and this status will be false if the communication is not yet completed, and true if it is. On the receiver side, completion again means that the message is received in the local buffer, after which a 'done' signal is issued (Q_{done}). After the Receiver has received the done signal, testing the status of the communication will result in a true state. Because the communication is synchronous, it is completed on the sender side once the final acknowledgement has been returned. The rest of the communication is comparable to the previous case (see Figure 24). However, a small detail is different. Because the Sender starts first this time, it immediately begins to send the first packet to the receiver. However, the Receiver is not yet ready to receive a message, and a non-acknowledgement is returned to the Sender. After some time packet 1 is sent for a second time. Now the Receiver already issued a receive command and is

ready to receive the message. The packet is copied into a local buffer and an acknowledgement returned to the Sender. Again, many other protocols are possible.

Finally consider asynchronous communication. We assume a non-blocking asynchronous send and a non-blocking receive, with the Sender first issuing the send command, see Figure 26.

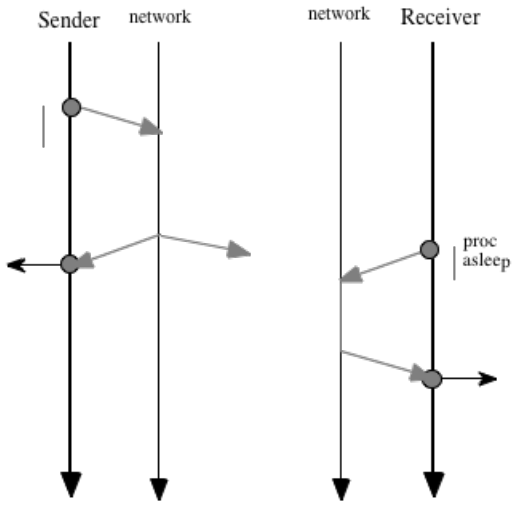


Figure 26: Feynman diagram for non-blocked asynchronous send and non-blocked receive, with the Sender first

Because of the asynchronous nature of the send, the Sender just copies the message into the network, and after the second packet is shipped out, the done signal is delivered after which the Sender has completed the communication. The packets are stored somewhere in the network (denoted by the zigzag lines) and after the Receiver is ready to receive the message, the packets are copied from the network into the local buffer.

2.6.3 Performance Issues

Let us assume that a point-to-point communication between a Sender and a Receiver can be modeled by two phases: setting up of the communication (i.e. preparing buffers, notifying the network, etc.) and a sending phase, in which bytes are actually moved from the Sender to the Receiver. Furthermore, assume that the set up time is constant and that the sending time depends linearly on the length of the message. The communication time for a point-to-point communication, T_{pp}, can now be written as

$$T_{pp} = \tau_l + n\tau_s \,, \qquad [19]$$

with τ_l the latency (usually measured in μs), τ_s the sending time (measured in μs/byte) and n the message length in byte. The transmission rate of the communication is just $1/\tau_s$ and is measured in Mbyte/s. Note that the communication channel can only send messages at the transmission rate for very large messages (try to prove this yourself). The linear model, Eq. [19], is very simple and in most cases the details of T_{pp} are more complicated (e.g. due to buffering). However, in many cases the linear model allows us to capture the main features of point-to-point communication and to understand the trade-off between latency and sending data.

Assume that the processors in a parallel computer have a computational speed of 30 Mflop/s. Each processor executes 30 operations per μs. If we also assume that the network latency is 3 ms (a typical value in a LAN), then each processor could execute 90000 operations in the time needed to start up a communication. Using this data one concludes that in order to get any benefit from parallel or distributed processing, sending one message should reduce the complexity of the problem by 90000 operations! This

means that programmers really need to think carefully about algorithms, if the goal of parallel computing is speeding up applications. The ease of parallel computing is inversely proportional to the latency. Higher latency means that more sophisticated algorithms need to be used to get any speedup. Better programming environments strive to reduce latency and make programming simpler. Also, the larger the latency, the larger the required granularity in order to get any useful speedup.

The same argument applies to the transmission rate. Assume that the transmission rate is 1 Mbyte/s. It then takes 4 μs to send one single precision floating point number over this network. In the same time, a processor could execute 120 operations. This means that in order to get any speedup, shipping 4 bytes from one processor to another should reduce the amount of needed operations by 120. The ease of parallel programming is proportional to the transmission rate. High transmission rates require less sophisticated algorithms to get good speedups.

Note that these arguments always refer to the communication speed relative to the computational speed. Compare this with the Eq. 18.c for the fractional communication overhead in the vector addition example. In fact, using Eq. 19 you should now be able to refine the equation for the fractional communication overhead, and really see above arguments in action. Try this !

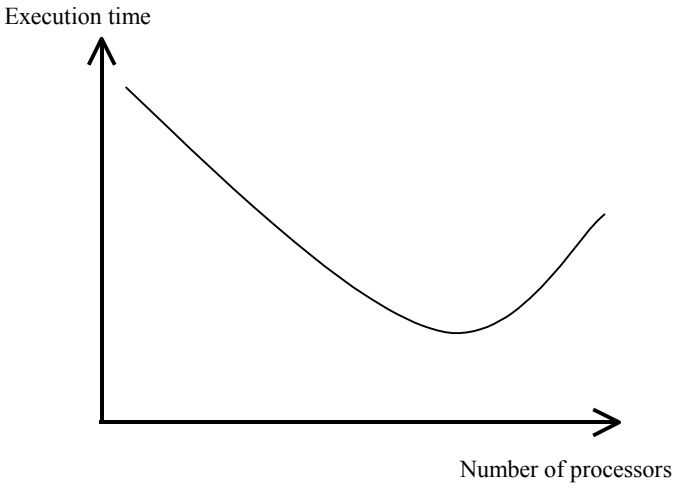


Figure 27: Execution time as a function of the number of processors for a typical parallel program.

Before moving to the topic of global communication let us pause for a while, and present some facts based on experience in parallel computing, which you should be able to understand and interpret by now. For every given parallel program executed on a real parallel computer there is an optimal number of processors at which the execution time is minimized. If more processors are added, the execution time increases again, as in Figure 27. This behavior is due to the overheads. If the number of processors is increased, usually the communication overhead increases. In the vector addition example the latency would not cause a constant communication time, but an increasing communication time if the amount of processors would be increased. This latency overhead would be the cause of the curve in Figure 27. Other overheads, such as e.g. I/O or pure sending time, may be the cause of the general shape. Now we must ask the question how large this optimal number of processors would be. If we have bad luck, this optimum is very small and massively parallel processing is just not possible because of this scaling behavior. This question cannot be answered in general, and highly depends on the specifics of the parallel algorithm and the parallel computer. However, another common experience, that restores the hope that massively parallel computing is feasible, is the fact that if we would measure the optimal number of processors as a function of the size of the computational problem (e.g. the length of the vectors in the parallel vector addition), this optimum number of processors increases with the problem size, as in Figure 28.

This is good news, if we scale the problem size with the amount of processors, we may be able to obtain high efficiencies for a large number of processors. As we started of with explaining that we need such massively parallel computers for these very large grand challenge applications, it seems that this scaling will allow us to reach our goals. The issue of scalability, which as we understand it now also includes scaling the workload, will be discussed in detail in section 4.

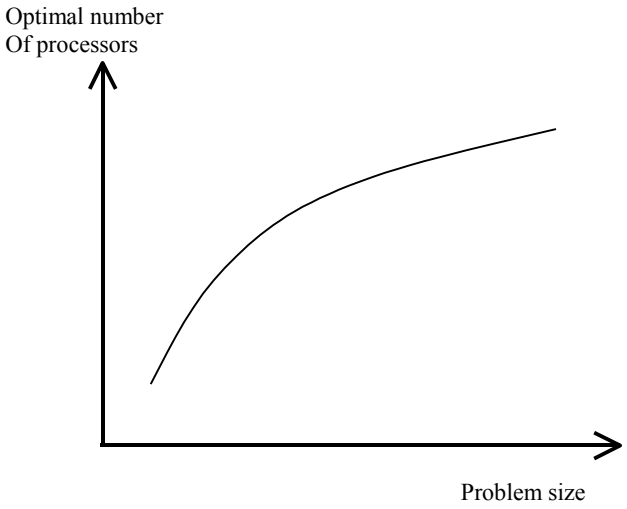


Figure 28: Optimal number of processors that minimizes the parallel execution time as a function of problem size.

2.6.4 Global Communication

So far we only considered point-to-point communication, i.e. one processor sending data to another. However, in many practical applications it turns out that global communication is required. Examples are the case that one processor sends a data item to all other processors, or when all processors should communicate with all others (an all-to-all communication). Therefore, besides basic point-to-point operations, parallel programming environments also offer global communication routines. This has two advantages. First, the application programmer does not need to develop this himself, and secondly, you may hope that message passing libraries contain optimized global communication routines.

We will discuss the four most common global communication routines, i.e. broadcast, concatenate, combine, and exchange. Note that these operations may have different names in different programming environments, and that some of them may be merged into one powerful global communication routine. Nevertheless, the four routines that we introduce below are the ones that are most commonly encountered.

The *broadcast* operation (see Figure 29) is used when one processor wants to send data to more than one other processor. Typically, but not necessarily, the data is sent to all other processors. Broadcast is a quite common operation in parallel programs. Clearly it is easy to implement a broadcast yourself using only point-to-point operations. However, it is easier and more flexible to have a single, maybe optimized broadcast operation available as a library routine.

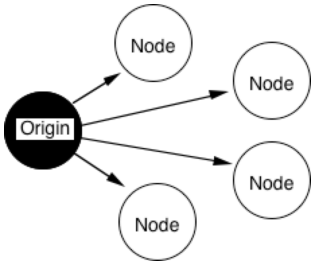


Figure 29: The broadcast operation.

The concatenate operation can be viewed as a set of parallel broadcast operations. The concatenate operation is performed when nodes each have parts of the data. After this call every node which is part of the operation will have a copy of the entire data set stored in its memory. Assume that the call will be made on 4 nodes and before the operation data is distributed as in Figure 30 on the left. After the concatenate operation the data distribution will look as in Figure 30 on the right. This operation can be encountered in for instance parallel matrix vector products, where the matrix and vectors are decomposed row-wise.

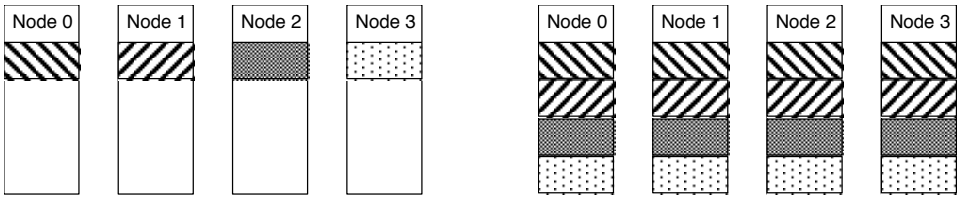


Figure 30: The concatenate operation, with on the left the data distribution before, and on the right after the operation.

The combine operation is an enhanced concatenation, in the sense that the full data set is not stored in each node, but an operation is performed on the full data set, resulting in one equal piece of data in all processors. After the combine operation every node that participates in the combine operation will have a copy of the data which is a result of applying a user specified function (e.g. addition) to the data stored in each node, including its own. Assume that the call will be performed on 4 nodes and that before the operation the data distribution is as in Figure 31 on the left. After the combine operation that data distribution will be as in Figure 31 on the right. The combine operation is used a lot, for e.g. calculation of a global sum, a global maximum or minimum, etceteras.

Figure 31: The combine operation, with on the left the data distribution before, and on the right after the operation.

Figure 32: The exchange operation, with on the left the data distribution before, and on the right after the operation.

In the exchange operation each node sends its own data to another node and at the same time receives data from yet another node. Usually the exchange is used to send data to a left neighbor and receive from a right neighbor. In that case, assume that the data before the exchange operation is distributed as in Figure 32 on the left. After the exchange operation the data distribution will be as in Figure 32 on the right. How would you implement such a right shifting exchange, using only blocking or only non-blocking

point-to-point communication? Beware of deadlock, i.e. a situation that all processors wait for input, and nothing happens.

2.6.5 The Message Passing Interface, MPI

Over the years many parallel programming environments for SPMD parallelism with message passing have been developed. These systems were originally strongly coupled to specific machines and vendors, e.g. the Parix environment for Parsytec, the CSTools environment for Meiko. These early systems were not portable and also contained just a small subset of all required functionality. An early message passing system that was available on many platforms, ranging from networks of workstations to massively parallel systems was Express (see e.g. Ref. 30, section 5). This system proved that it is possible to write efficient and portable message passing parallel programs. The real breakthrough came with PVM, the Parallel Virtual Machine [34]. This system was originally developed as a message passing system that could support heterogeneous distributed computing, i.e. parallel programs that execute on a heterogeneous network of workstations. Around 1992 PVM became a de-facto standard, in the sense that many vendors supported PVM on their parallel machines and many applications programmers developed message passing programs in PVM. For the first time source level portability of message passing programs written in PVM was available over a large range of parallel computers.

PVM was never developed as a standard for message passing SPMD programs and therefore it suffered from many deficiencies. This was realized by most research groups and vendors of parallel computers, and during the SuperComputing 1992 conference, a forum was established that would define an industrial standard for SPMD message passing programs, to be called The Message Passing Interface or MPI. The MPI forum was open to everybody, and after one and a half years of hard working version 1 of the MPI standard was officially announced in May 1994. At the same time an implementation of the MPI standard was freely available in the public domain. This was the celebrated MPICH system. Very soon after the release of the standard and of MPICH, many vendors made MPI available for their own systems (usually their implementation of MPI was based on MPICH) and within 2 years MPI became the true standard for message passing SPMD programming. Currently, MPI 1.1 is available on most systems. For more information on MPI, best is to start looking at the MPI web site, http://www.mcs.anl/gov/mpi/index.html. In these lecture notes we give a small introduction to MPI. More detailed information is available in the notes that accompany the lab course.

MPI is an interface for development of SPMD parallel programs using message passing. It is available on most parallel computers, and on (heterogeneous) clusters of workstations. It provides source code portability between different parallel computers and can be considered as the current industrial standard. A parallel MPI program consists of a number of parallel processes, communicating through calls to the MPI libraries. The processes are loaded on the processors of a parallel computer. From the point of view of MPI, all parallel processes are fully connected, i.e. they can all communicate directly with each other. In most cases the underlying processor network is not fully connected, and the low level communication libraries will take care of routing data to the correct destinations. Each process has its own process identification number. An entire parallel job is started by simply invoking e.g.

```
mpirun -np 4 foo
```

which starts 4 copies of the program `foo`. If 4 or more processors are available, each copy is started on a different processor.

We will now go through an example of a parallel matrix vector product, and we will end with a fully working MPI program. We will multiply a matrix with N rows and M columns by a vector with M entries. This program will operate on vectors of any size, M, and runs on N processors. Each processor receives one row of the full matrix, see Figure 33. Of course, this is a real restriction. This parallel matrix vector product is meant to demonstrate some features of MPI, and is certainly not meant to show how one should do a parallel matrix vector product. For that you should take a look in Refs. [30, 31].

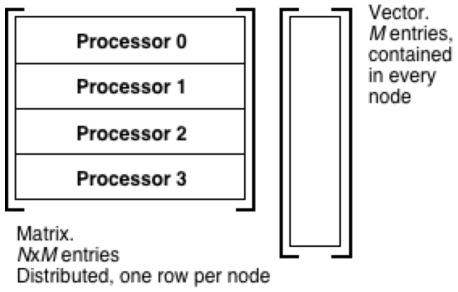


Figure 33: The parallel matrix vector product, with a single row decomposition of the matrix.

The basic program outline is as follows:

- Find how many processes are available and use this value as *N*, the number of rows of the matrix;
- Process 0 prompts for the value of *M*, and reads the vector;
- The vector is broadcasted to all other processes;
- Process 0 read the matrix elements, and distributes the rows to the other processes;
- All processes perform an inner product of the matrix row and the vector and send the result back to process 0;
- Process 0 prints the results.

We will first show some isolated parts of the MPI program, and finally the full program.

First, MPI must be initialized with the `MPI_Init` routine, and at the end of the program, must be stopped again using `MPI_Finalize`. If an error occurs somewhere in the program, one can use `MPI_Abort` to exit. For examples, see below.

```
#include <mpi.h>
main(int argc, char *argv[])
{
/* initialize MPI */
MPI_Init (&argc, &argv)

/* lots of useful code */

if (error)
  MPI_Abort (MPI_COMM_WORLD, error) ;

/* more code */

/* finalize MPI */
MPI_Finalize ();
}
```

The `MPI_COMM_WORLD` variable above is a so-called communicator, which is a grouping of processes. Here we will not further discuss this very useful feature of MPI and limit ourselves by remarking that the `MPI_COMM_WORLD` communicator contains all processes that we started with `mpirun`. Within a communicator, each process has its own identification, which is called *rank* in MPI. Furthermore, each communicator has a certain number of processes. Using the default `MPI_COMM_WORLD` communicator, one can find the number of parallel processes using `MPI_Comm_size` and the rank using `MPI_Comm_rank`, see below.

```
/* get the context, i.e. the number of
   processors and their rank. */
```

```
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &myid);
```

Point to point communication in MPI is only between processes in the same communicator. So, within the `MPI_COMM_WORLD` communicator all processes can communicate with each other. The basic synchronous blocking send and receive routines are

```
Int MPI_Send (void *buff, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
Int MPI_Recv (void *buff, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status)
```

MPI contains many more modes of point-to-point communication. The `*buff` points to a buffer where the data that must be sent is located or where the received data must be stored. All messages in MPI are typed. The standard types include

```
MPI_Char     signed char;
MPI_INT      signed int;
MPI_FLOAT    float;
MPI_DOUBLE   double.
```

One can also define derived datatypes such as strided vectors or very general structures. The variable `count` tells the send or receive operation how many data items of `MPI_Datatype` should be sent or received. The `dest` variable in the send operation is the rank of the receiving process, and the `source` variable of the receiving process is the rank of the sending process. The `tag` variable can be used to distinguish messages from each other.

In our parallel matrix vector product we use the point-to-point communication to collect the results of the matrix vector product in process 0. Below we show how this is accomplished. Process 0 first stores it's own result in the result vector, and next receives from all other processes their results. All other processes send their own result back to process 0. Based on their rank, process 0 knows which row of the matrix the other processes received, and process 0 knows therefore where to store the result. The code is shown below.

```
/* all processors have calculated their partial result, stored
it in a variable res. Now, each processor sends it to processor
0, which stores it in a vector resvector */

if (myid == 0) {
  /* put own result in vector */
  resvector[0] = res;

  /* receive results from other processors */
  for (i=1, i<numprocs; i++) {
    MPI_Recv (&resvector[i], 1, MPI_FLOAT, i,
              i, MPI_COMM_WORLD, &status);
    }
}
else {
  /* send results back to processor 0 */
  MPI_Send (&res, 1, MPI_FLOAT, 0, myid,
            MPI_COMM_WORLD);
}
```

A global broadcast operation is used to send the argument vector from process 0 to all other processes, as shown below. Note the calling of the `MPI_Bcast` routine. All processes issue exactly the same command. The 0 in the argument list means that the

process with rank 0 is the originating process, and all other processes in the communicator (in this case `MPI_COMM_WORLD`) will receive the data. The data is stored in the buffer `vector`, and a total of `size` floating point numbers are broadcasted from process 0 to all other processes.

```
/* after processor 0 has read the argument vector, with its
length stored in the variable size, it must be broadcast to all
other processors */

MPI_Bcast (vector, size, MPI_FLOAT, 0, MPI_COMM_WORLD)
```

We are now ready to show the full MPI program. Study it in detail and make sure that you understand it completely.

```
#include <stdio.h>
#include <mpi.h>
#define MAXVEC 20 /* size of biggest vector */

main(int argc, char *argv[])
{
   int i, j, numprocs, myid, size;
   float vector[MAXVEC], resvector[MAXVEC]
   float matrix[MAXVEC][MAXVEC];
   float res;
   MPI_Status status;

   /* Initialize MPI */
   MPI_Init (&argc, &argv);

   /* Get environment, i.e. the number of processors */
   /* and the i.d. */
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   if(numprocs > MAXVEC) {
      if(myid = 0)
         printf("to many processes, maximum allowed
                 %d/n",MAXVEC);
         MPI_Abort(Exit_Failure);
   }

   /* Read the vector and matrix, by the process with i.d. 0 */
   if(myid == 0) {
      printf("Enter size of the vector : ");
      scanf("%d",&size);
      while(size > MAXVEC) {
         printf("to large, maximum allowed value is %d\n
                 try again : ", MAXVEC);
         scanf("%d",&size);
      }

      /* read vector */
      printf("Enter vector values :\n");
      for(i=0; i<size; i++) scanf("%f",&vector[i]);

      /* read matrix */
```

```
      printf("Enter %d x %d matrix (row wise)
              :\n",numprocs,size);
      for(i=0;i<numprocs;++i)
         for(j=0;j<size;j++)
            scanf("%f",&matrix[i][j]);
      }

   /* broadcast the size of the vector from process 0 */
   /* to all other processes */
   MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);

   /* broadcast the vector from process 0 to all other */
   /* processes */
   MPI_Bcast(vector, size, MPI_FLOAT, 0, MPI_COMM_WORLD);

   /* send rows of the matrix from process 0 to other */
   /* processors */
   if(myid == 0) {
      /* send rows of the matrix */
      for(i=1;i<numprocs;i++)
         MPI_Send(matrix[i], size, MPI_FLOAT, i, i,
                  MPI_COMM_WORLD);
      }
   else {
      /* receive a row from process 0 */
      MPI_Recv(matrix[myid], size, MPI_FLOAT, 0, myid,
               MPI_COMM_WORLD,
      &status);
      }

   /* now, finally, do the matrix vector product in parallel */
   /* Each process simply performs an inner product.         */
   res = 0.0;
   for(i=0;i<size;i++) res += vector[i]*matrix[myid][i];
   /* all processes send their result back to process 0, */
   /* which puts the results in the result vector and */
   /* prints the results.*/
   if(myid == 0) {
      /* put own result in resvector */
      resvector[0] = res;

      /* receive results from other processors */
      for(i=1;i<numprocs;i++)
         MPI_Recv(&resvector[i], 1, MPI_FLOAT, i, i,
MPI_COMM_WORLD,
          &status);

      /* print results */
      printf("\n\nthe matrix was ..\n\n");
         for(i=0; i<numprocs; i++) {
            for(j=0; j<size; j++)
               printf("  %f ",matrix[i][j]);
            printf("\n");
         }
```

```
      printf("\n\nthe input vector was ..\n\n");
      for(i=0; i<size; i++)
         printf("  %f\n",vector[i]);
      printf("\n\nthe result vector is ..\n\n");
      for(i=0; i<numprocs; i++)
         printf("  %f\n",resvector[i]);
   }
   else {
   /* send results of innerproduct back to process 0 */
      MPI_Send(&res, 1, MPI_FLOAT, 0, myid, MPI_COMM_WORLD);
   }

   /* Terminate MPI */
   MPI_Finalize ();
}
```

3 A Case Study, the Guitar String

3.1 Introduction

It is now time to study a case of parallelizing a real application. We choose a relatively simple model of vibrations in a guitar string, and will go through all phases in modeling and simulation, as drawn in Figure 17. The main focus will be on the decomposition and the resulting parallel program. Yet, we also need to go through the modeling and solver layer in some detail in order to see an example of how these phases actually work and how the mapping from one layer to another is accomplished. An interesting question is if you are able to keep inherent parallelism, that may be present in the original application, preserved down to the parallel virtual machine layer.

3.2 The model

Imagine a guitar player who wonders how it is possible that strumming the strings of his guitar results in the beautiful guitar sound. Strumming a string results in vibrations in the string, that are coupled to the body of the guitar. The body also starts vibrating and together with the string induces air pressure fluctuations that we recognize as the sound of a guitar. Building a full model of this would be very complicated, and before embarking on such a journey, on usually starts with taking the full system apart and trying to understand the behavior of the parts. In this case, it is clear that the basic phenomenon to be studied is the vibration of the guitar string. Therefore, we decide the study an isolated guitar string of length L, held at a tension T, and fixed at its end points. We will only consider transverse (i.e. perpendicular to the string) vibrations (see Figure 34).

More specifically, we seek a model for the amplitude $y(x,t)$ of the string as a function of the position x along the string, with $0 < x < L$, and as a function of the time t. We will first derive an equation for the amplitude $y(x,t)$.

In equilibrium the amplitude of the string is zero everywhere and the tension is equal to T everywhere. Next, we assume that as the string vibrates, it has a small amplitude y, such that the tension remains equal to T everywhere in the string. From Figure 35 we will evaluate the tensions on a small part of the string, on point x and $x + dx$, where dx is very small.

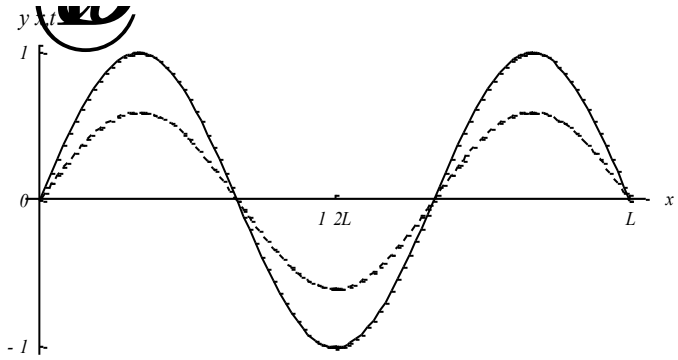


Figure 34: The vibrating string, two solutions are shown on different times t.

Because of the curvature of the string, the tensions on both points are not exactly equal. The difference in the *y*-components of tension on position *x* and *x* + *dx* yields a resulting force, that can be expressed as

$$F_y = T\left(\sin(\alpha') - \sin(\alpha)\right). \qquad [20]$$

Because we assumed that the amplitude *y* is very small, the angles α and α' are also very small. Therefore, $\sin(\alpha) \approx \tan(\alpha)$. We immediately recognize $\tan(\alpha)$ as the slope the string, i.e. $\tan(\alpha) = \partial y / \partial x$. Using all this, we rewrite Eq. [20] to

$$F_y = T\left(\left.\frac{\partial y}{\partial x}\right|_{x+dx} - \left.\frac{\partial y}{\partial x}\right|_{x} \right), \qquad [21]$$

which, in the limit of very small *dx* becomes

$$F_y = T \frac{\partial^2 y}{\partial x^2} dx. \qquad [22]$$

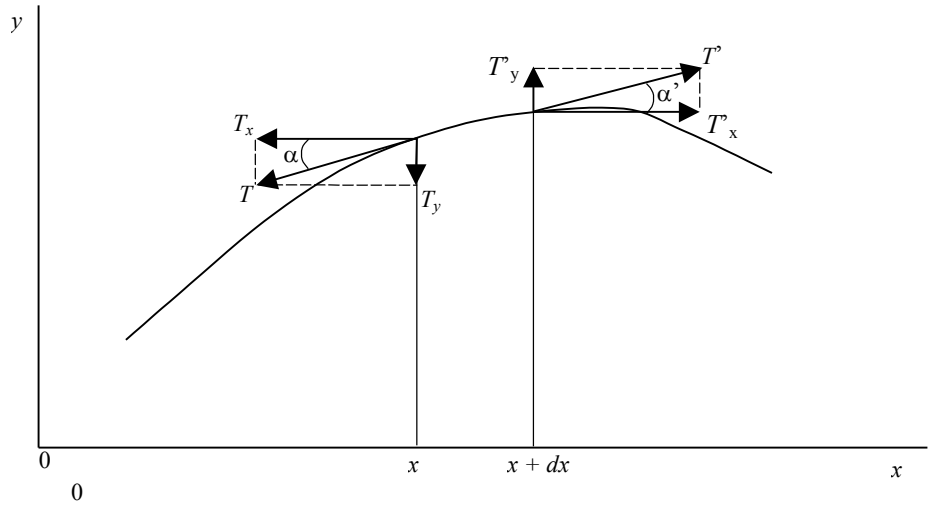


Figure 35: Tensions on a part of the string.

The length of the string between coordinates *x* and *x* + *dx* is *ds*. In the limit of very small *dx*, we find that *ds* = *dx*. If we call μ the mass of the string per length unit, Newton's law results in

$$F_y = \mu dx \frac{\partial^2 y}{\partial t^2}. \qquad [23]$$

By combining Eq [22] and [23] we find the one-dimensional wave equation,

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2}, \qquad [24]$$

where

$$c = \sqrt{T/\mu} \qquad [25]$$

is the velocity of sound in the string.

Before we proceed, note that in Figure 35 the resulting force in the *x*-direction is zero, which means that the movements of the string are only in the *y*-direction. Prove this yourself.

The one-dimensional wave equation is our model of vibrations in the string. However, before we can solve it, we need to supply boundary – and initial conditions. The boundary conditions have already been mentioned. The string is fixed on both ends, or, mathematically,

$$y(0,t) = y(L,t) = 0. \qquad [26]$$

The initial condition gives the amplitude of the string on time $t = 0$, i.e.,

$$y(x,0) = f(x), \qquad [27]$$

where $f(x)$ can be any smooth function that satisfies Eq. [26].

Although this model looks very simple, it is actually used in studies of vibrating guitar and piano strings. If you are interested you could read Ref. [35] as a very nice introduction in this field.

The one-dimensional wave equation [24] with boundary condition, Eq. [26], and initial condition, Eq. [27], can be solved analytically. We will not derive the result, which is easily obtained using Fourier analysis, but just give the final solution,

$$y(x,t) = \sum_{m=1}^{\infty} B_m \sin\left(\frac{m\pi x}{L}\right)\cos\left(\frac{m\pi ct}{L}\right), \qquad [28]$$

where the coefficient B_m is determined by the initial condition as

$$B_m = \frac{2}{L}\int_0^L f(x)\sin\left(\frac{m\pi x}{L}\right)dx. \qquad [29]$$

Each term in the series in Eq. [28] must be interpreted as a normal - or eigen mode of the string. The $m = 1$ mode is the ground tone of the string. It has it's maximum in the middle of the string, and vibrates with a frequency

$$\nu = \frac{1}{2L}\sqrt{T/\mu}. \qquad [30]$$

Increasing the tension in the string or taking a lighter string results in a higher frequency of the ground tone. If you are a guitar player you know this by experience. The string also supports higher harmonics, whose frequencies are m times that of the ground tone. The higher harmonics have so-called knots, places where the amplitude is always zero. For example, the $m = 2$ harmonic has a knot for $x = L/2$. The subtle mix of ground tones and higher harmonics gives each string instrument it's characteristic sound.

3.3 The Solver

We have solved our model, i.e. the one-dimensional wave equation, and could proceed by analyzing in detail what would happen if we take special initial conditions that resemble, e.g., picking a string. However, we proceed in a different way, and seek a numerical method to solve the wave equation. In doing so we enter the field of numerical analysis. We will only touch upon some issues from this field. If you want more information, and like to have it in a cook book style you are advised to read Numerical Recipes [36]. For more in-depth discussion Ref. [37] could be consulted.

We have to cast the wave equation into a form that allows simulation on a computer. We do this by *discretizing* the equation. First, we define a computational grid, see Figure 36. We only seek solutions for $y(x_i,t_j)$, where $x_i = i\Delta x$ and $\Delta x = L/N$, where N is the number of points on the x-axes. Furthermore, $t_j = j\Delta t$, where Δt is a predetermined time step.

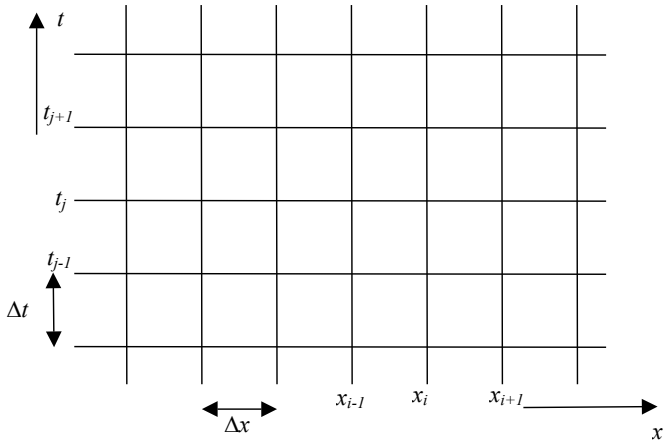


Figure 36: The computational grid.

Now recall that the formal definition of the derivative of a function $g(x)$ is

$$\frac{\partial g(x)}{\partial x} = \lim_{dx \to 0} \left(\frac{g(x+dx) - g(x)}{dx} \right).$$

If the grid spacing Δx is small enough, we may write to a good approximation

$$\frac{\partial g(x)}{\partial x} \approx \frac{g(x+\Delta x) - g(x)}{\Delta x}. \qquad [31]$$

Eq. [31] is called a finite difference representation of the derivative. The approximation in Eq. [31] gets better if Δx becomes smaller. Mathematically we say the error in Eq. 31 is of order Δx^2, or $O(\Delta x^2)$. This is easy to prove by considering the Taylor expansion of $g(x+\Delta x)$ around x. We apply Eq. [31] to discretize the wave equation. First consider the time derivatives in the wave equation:

$$\frac{\partial y(x_i, t_j)}{\partial t} \approx \frac{y(x_i, t_j + \Delta t) - y(x_i, t_j)}{\Delta t}. \qquad [32]$$

Using Eq. [32] we can now derive a finite difference equation for the second time derivative,

$$\frac{\partial^2 y(x_i,t_j)}{\partial t^2} \approx \frac{y(x_i,t_j-\Delta t)-2y(x_i,t_j)+y(x_i,t_j+\Delta t)}{\Delta t^2}. \qquad [33]$$

A comparable result is easily obtained for the second derivative in x,

$$\frac{\partial^2 y(x_i,t_j)}{\partial x^2} \approx \frac{y(x_i-\Delta x,t_j)-2y(x_i,t_j)+y(x_i+\Delta x,t_j)}{\Delta x^2}. \qquad [34]$$

Combining Eqs. [24, 33, 34] results in

$$\begin{aligned} y(x_i,t_j+\Delta t) = {} & 2y(x_i,t_j)-y(x_i,t_j-\Delta t) \\ & +\tau^2\left(y(x_i-\Delta x,t_j)-2y(x_i,t_j)+y(x_i+\Delta x,t_j)\right) \end{aligned} \qquad [35]$$

where

$$\tau = \frac{c\Delta t}{\Delta x}. \qquad [36]$$

Equation [35] is an expression to calculate the amplitude on a next time step $t_j+\Delta t$, using values on earlier times (i.e. on t_j and $t_j-\Delta t$). This is called an explicit finite difference scheme. Many other schemes to discretize the wave equation can be devised, e.g. implicit finite difference equations, but we pick this one because it is relatively easy to parallelize. However, before moving to this subject we do need to say a few more words about the numerical scheme that we just derived. For all details we again refer to books on numerical algorithms, see e.g. Refs [36, 37].

Explicit versus Implicit

Equation [35] is called explicit because it gives us a formula to calculate explicitly the amplitudes on a next time step, i.e. without the need to solve equations. Alternatively, one could have an implicit finite difference scheme. Here, in order to calculate the amplitudes on the next time step, one must solve a system of equation. This happens when e.g. we would estimate the spatial (in the x-direction) derivatives on the next time step, i.e. on $t + \Delta t$, instead of on time t. The resulting finite difference equation would now contain the amplitudes on x_i on the next time step, but also on $x_i \pm \Delta x$. Writing down the equations for all values of i results in a set of linear equations that need to be solved. The main advantage of implicit schemes is that they are unconditionally stable, and therefore allow much larger time steps then explicit schemes, however at the expense of the need to solve systems of equations.

In general, we judge the quality of a numerical scheme on the basis of the following three criteria:

- consistency,
- accuracy,
- stability.

It would go to far to discuss these items in detail. We will just explain what is meant by these criteria, and shortly state how they apply to our finite difference scheme in Eq. [35].

Consistency is the requirement for any algebraic approximation to a partial differential equation to reproduce the partial differential equation in the limit of an infinitesimal time step and grid spacing. It is easy to prove that our finite difference scheme is consistent with the wave equation.

A natural requirement of the finite difference scheme is that the computed amplitudes resemble, i.e. are very close, to the real solutions of the original wave equation. Accuracy and stability are related to this requirement. Accuracy is concerned with local errors. Local errors arise from two sources. First are the roundoff errors resulting from the finite word length of numbers within a computer, and second are the truncation errors caused by representing continuous variables by discrete sets of values. Generally, roundoff errors are much smaller than truncation errors, and provided the scheme is stable (see below)

they can usually be ignored. Truncation errors, which arise from the representation of continuous quantities by discrete sets of values are usually described in terms of the difference between the differential and algebraic equations. A measure of the smallness of truncation errors is given by the order of the difference scheme.

Stability is concerned with the propagation of errors. Even if truncation and roundoff errors are very small, a scheme will be of little value if the effects of the small errors grow rapidly with time. Instability arises from the non-physical solutions of the discretized equations. If the discrete equations have solutions which grow much more rapidly than the correct solution of the differential equations, then even a very small roundoff error is certain eventually to seed that solution and render the numerical results meaningless. A numerical method is stable if a small error at any stage does not lead to a larger cumulative error. Without prove we say that our finite difference scheme is stable if and only if

$$\tau \leq 1 . \qquad [37]$$

This equation is a so-called Courant stability condition. This stability condition in facts limits the time step that we can take to $\Delta t \leq \Delta x/c$. We choose Δx in such a way that we have a high enough resolution to "support" certain wavelengths on the grid. As a rule of thumb, to support a wave with a wavelength λ we should choose $\Delta x < \lambda/10$. This means that in order to support the $m = 1$ node, with $\lambda = 2L$, we should take $N > 5$. In general, to support solutions up to mode m, $N > 5m$. In combination with the sound speed c in the string this determines the maximum time step Δt that we are allowed to take in the numerical scheme. Of course we would like to take an as large as possible time step. However, in the explicit finite difference scheme this is not possible because of the stability constraint. So-called implicit schemes, as discussed in the sidebar, are unconditionally stable, which means that for any Δt stability is guaranteed. This allows to take much larger time steps, however at the expense of the need to solve a set of equations. Also it is important to note that explicit schemes are easy to parallelize, whereas this is much more difficult for implicit schemes.

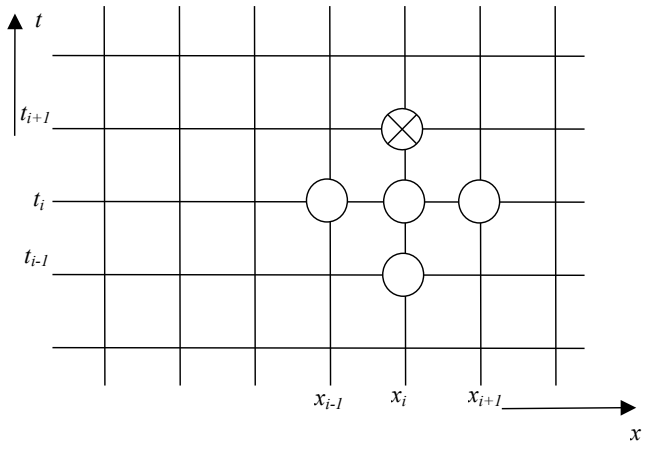


Figure 37: The stencil of the finite difference equation. The circle with the cross inserted is the point that is going to be updated, the empty circles are the points that are needed to perform the upate.

The explicit finite difference formula, Eq. [35] is a so-called stencil operation. In order to calculate the amplitude on a next time step we need a small number of points on previous times to perform the calculation. This set of points is called the stencil, which is drawn in Figure 37. The important property of such stencil operations is that we only need a small and *local* neighborhood to update the amplitude. This locality is very important for parallelization.

3.4 Parallelization

The finite difference equation is parallelized by means of a decomposition of the computational domain. Our computational domain is formed by the (x, t) plane, as in Figure 36. In principle, we have many possibilities to decompose this grid (remember the wall building example in section 2.5.3). However, because we deal with an initial value problem the only choice is to do a spatial decomposition, i.e. decomposition in the x-direction. In this way, each processor gets a part of the string, and simulates the time behavior of this specific part, see Figure 38. We could also have chosen for a temporal decomposition, i.e. splitting up the computational grid in the t-direction, and giving periods of time to each processor. Why is this not a good idea, and can you think of a situation where such decomposition may be preferable?

Due to the locality of the stencil (see Figure 37) this spatial decomposition leads to the identification of *internal* and *boundary* points, see Figure 39. An internal point is a point whose stencil lies entirely in the domain assigned to a processor. Each processor can update its internal points fully in parallel, without the need to exchange information with other processors. Boundary points are defined as those points where a part of the stencil lies in the domain of another processor. To update boundary points it is necessary to get some information from another processor. The exact size and shape of the stencil determines which points are internal or boundary points. In our example the most left and right points in the domain of a processor are boundary points. In many realistic three-dimensional applications, such explicit finite difference equations with local stencils occur, and therefore our example of the one-dimensional wave equation, solved using explicit finite differencing, is a relevant and important case.

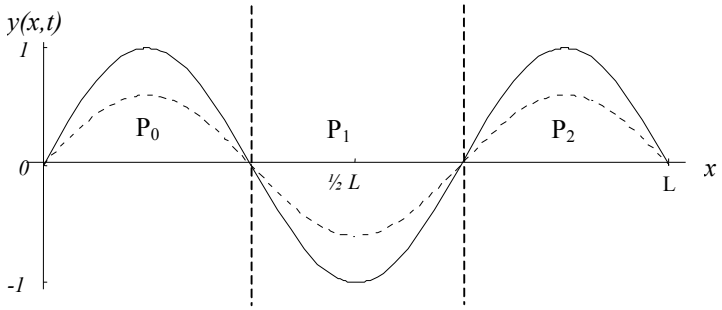


Figure 38: Spatial decomposition of the computational domain, in this example for three processors, P_0, P_1, and P_2.

We can now write down the basic parallel algorithm. Assume that the processors are configured as a linear array, i.e. each processor has a left and right neighbor, except the processor on the far left and on the far right of the chain. Mapping of the decomposition to the processor network is trivial. All processors start to initialize the starting values, and next start the iteration. In each pass of the iteration, the processors first exchange the end nodal values with their left- and right neighbors, using an exchange operation as discussed in section 2.6.4. Now all processor have all information available to update from $y(x_i, t_j)$ to $y(x_i, t_j + \Delta t)$ for all j. Below we show the pseudo code of the full parallel program that will be used in the lab course that accompanies this lecture. This pseudo code is provided without further explanation.

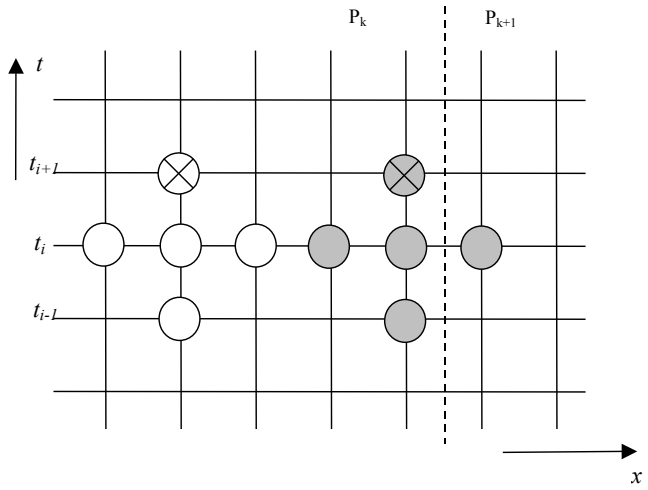


Figure 39: Internal (crossed white) and boundary (crossed gray) points in a spatial domain of processor P_k. The corresponding stencils are also drawn.

```
proc_begin main ()
   [MPI_Init- initialize MPI]
   [ring - set up ring topology; assign nearest neighbors]
   for_begin (interactive command loop)
      [scanf - read command from input stream]
      if_begin (command in INITIALIZE) then
         proc_call initialize_problem (assign starting values)
      else_if (command is UPDATE) then
         [scanf - read number of steps, k, from input stream]
            proc_call do_steps (perform k update steps)
      else_if (command is OUTPUT) then
         proc_call output_results (write results to output using
          printf)
      else_if (command is STOP) then
         [terminate program]
      if_end
   for_end
proc_end

proc_begin do_steps (perform k update steps)
   for_begin (a total of k loops)
      [load end values into l_neighbor and r_neighbor]
      [send r_neighbor to processor on right while receiving
       into l_point form processor on left]
      [send l_neighbor to processor on left while receiving into
       r_point from the processor on right]
      for_begin (number of points in subdomain, i)
         if_begin (the point is a global endpoint) then
            [the value remains zero]
         else_if (the point is a left boundary) then
            [use l_point in place of ψi-1 to do update]
         else_if (the point is a right boundary) then
            [use r_point in place of ψi+1 to do update]
         else_if (otherwise) then
            [perform the usual update]
         if_end
```

```
        for_end
        [replace previous and current step values for next step]
      for_end
    proc_end
```

3.5 Performance Analysis

In this section we will perform an analysis of the performance of the parallel string simulation. We will only investigate the performance of the inner kernel, i.e. the parallel execution time of one pass of the parallel iteration. We do this to give you a taste of performing such analysis. However, you should realize that in general it is not completely fair to only look at the inner kernel, because many overheads that may reduce the performance of our parallel program are outside the inner loops. You could try, after reading this section, to set up your own, more involved performance analysis that takes these issues also into account.

Define τ_{calc} as the time needed to update one internal point and τ_{comm} as the time needed to exchange one point between neighboring processors. The parallel execution time of one iteration of the parallel string simulation, as a function of the number of processors p and N, is given by

$$T_P(N) = \frac{N}{p}\tau_{calc} + 2\tau_{comm}. \qquad [38]$$

The relative efficiency is

$$\varepsilon = \frac{1}{1 + \frac{2p}{N}\frac{\tau_{comm}}{\tau_{calc}}}. \qquad [39]$$

Eq. [39] has the same form as Eq. [15]. The fractional communication overhead is

$$f_c = \frac{2p}{N}\frac{\tau_{comm}}{\tau_{calc}}. \qquad [40]$$

We see that if the grain size N/p is large compared to the relative cost of sending data, i.e. relative to τ_{comm}/τ_{calc}, f_c becomes very small. In that case the relative efficiency can be very close to 1.

In this calculation we ignored the fact that N/p is usually not exactly an integer. In that case we will induce a slight load imbalance, because some processors will receive 1 more grid point as compared to other processors. We can capture this load imbalance in a fractional load imbalance overhead, f_l, just like we captured the communication overhead in the f_c parameter.

Using the ceiling function, $\lceil x \rceil$, which rounds x to the smallest integer that is larger than x, e.g. $\lceil 1.3 \rceil = 2$, we write the parallel execution time of one iteration of the parallel string simulation in the following, more correct form,

$$T_p(N) = \left\lceil \frac{N}{p} \right\rceil \tau_{calc} + 2\tau_{comm}. \qquad [41]$$

Next, Eq. [41] is written as

$$T_p(N) = \frac{T_1(N)}{N}\left(1 + f_c + f_l\right), \qquad [42]$$

where $T_1(N) = N\ \tau_{calc}$, f_c is the fractional communication overhead of Eq. [40]. In this formulation the relative efficiency becomes $1/(1+f_c+f_l)$ and after some algebra the fractional load imbalance overhead f_l turns out to be

$$f_l = \frac{p\left\lceil \frac{N}{p} \right\rceil - N}{N}. \qquad [43]$$

Let us try to interpret Eq. [43]. First, look more closely at the fractional communication overhead. This should be interpreted as the fraction of total time lost in communication, as compared to the total amount of computation. We should also interpret the load imbalance in this way. The denominator is N, i.e. the total amount of work that must be done. The first term in the numerator is the total amount of work that could have been done by p processors, and the second term in the numerator is again the total amount of work. All in all, the numerator is the amount of *lost* cycles, in terms of work, due to the load imbalance, and the fractional load imbalance overhead therefore is the fraction of lost cycles relative to the total amount of work.

Suppose that the communication overhead is zero, that $p = 3$ and $N = 6$. In this case all processors have 2 grid points and there is no load imbalance, $f_l = 0$ and $\varepsilon = 1$. Next, assume that $N = 5$. In this case the first and second processor get 2 grid points and the third only gets 1 grid point. This means that processor 3 will be idle half of its time, while the other two processors are still working on their second grid point. The amount of lost cycles is 1, $f_l = 1/5$, and therefore $\varepsilon = 5/6$. If $N = 4$ the situation gets worse. In this case only processor 1 has two grid points, the other two processors have only one grid point. The total amount of lost work due to load imbalance now is 2, making $f_l = 2/4$ and $\varepsilon = 2/3$. Note that the influence of this form of load imbalance is very small if the grain size N/p is large. In that case f_l is getting small, usually much smaller than the fractional communication overhead, and can then be ignored. However, in other applications f_l might be of the same order of magnitude as f_c.

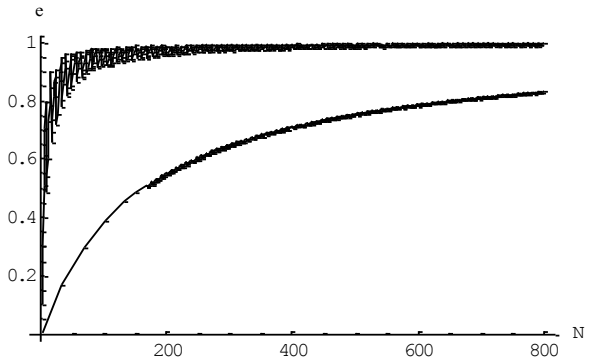


Figure 40: The relative efficiency of one iteration of the parallel string simulation, as a function of the number of grid points. The number of processors is 8, the upper curve is for $\tau_{comm}/\tau_{calc} = 0.1$ and the lower curve is for $\tau_{comm}/\tau_{calc} = 10$.

As an example, Figure 40 shows the relative efficiency of one iteration as a function of the number of grid points. The number of processors is 8. The upper curve shows the situation where the communication overhead is relatively small. Here we clearly see the influence of the load imbalance (the spikes on the curve). When is the computation completely balanced, at the top of a spike or at the bottom, and why? The lower curve shows the situation when the communication time is larger, resulting in a much smaller fractional load imbalance. In both cases, the efficiency increases with increasing grain size, i.e. with increasing N.

4 Performance Analysis and Benchmarking

4.1 Introduction

In this final section we will study in more detail the issue of performance analysis of a parallel program. First, we will encounter some important theoretical concepts, which are in fact related to the by now familiar speedup and efficiency. Next, we will study the question how one should measure, in a controllable way, the performance of parallel computers and parallel programs. This is the field of benchmarking.

In 1991, D.H. Bailey from NASA Ames published, more or less as a joke, 12 ways to fool the masses into believing that your parallel computer is the fastest of them all. Later, Bailey learned that such things actually happen. Therefore, it is very important to realize what it is exactly when you are quoting performance of a (parallel) computer. Below are Bailey's twelve ways [38]:

1 Quote 32-bit performance, not 64-bit results, and compare with others' 64-bit results,
2 Present inner kernel performance figures as the performance of the entire application,
3 Quietly employ assembly code and other low level constructs, and compare with Fortran and C implementation,
4 Scale up problem size with number of processors, but don't clearly disclose this fact,
5 Quote performance results linearly projected to a full system,
6 Compare your results against unoptimized, scalar, single processor code on vector machines,
7 Compare with an old code on an obsolete system,
8 Base MegaFlop operation counts on the parallel implementation, not on the best Sequential Implementation,
9 Quote performance in terms of processor utilization, parallel speedups or MegaFlop/s per dollar (peak MegaFlop/s -not sustained-),
10 Mutilate the algorithm used in the parallel implementation to match the architecture. In other words, employ algorithms that are numerically inefficient in order to exhibit artificially high MegaFlop/s rates,
11 Measure parallel run times on a dedicated system, but measure conventional run times in a busy environment,
12 If all else fails, show pretty pictures and animated video's, but DON'T talk about performance.

Is is easy to find examples where such tricks are utilized. In fact, we already did this. In the case of the vibrating string we studied performance of the inner kernel. Now look at item 2 of the list. In many cases performance of inner kernels are reported, without clearly telling this. Another popular one is item 5. Suppose you run your program on say 32 processors, and then multiply this figure by 4 and report that the result is the performance on 128 processors. This is a very bad thing to do, and you should already understand why. However, in many cases people are tempted to do this (e.g. when they have to convince their manager to buy an expensive system, it is better to show impressive performance figures).

Given all this it is very important to understand the basics of performance of parallel programs and to understand what it takes to measure and report performance of parallel computers in a scientifically correct way.

4.2 Performance Evaluation Metrics

4.2.1 Speedup and efficiency

Speedup and efficiency were introduced in section 2.5.4. These metrics are heavily used. One often sees graphs where the speedup or efficiency as a function of the number of processors is presented. Although very useful, these numbers should be interpreted with much care. It is e.g. very tempting to state that your parallel program that solves a certain problem has a larger speedup than that of your neighbor. Although this sounds very

impressive, in fact nothing has been said. Even worse, strictly speaking such a comparison should not be made. Why is this?

Speedup and efficiency are relative numbers. You compare the execution time of a parallel program with the execution time, on the same computer, of the parallel program executing on one processor (or, even better, with the execution time of the fastest sequential program). Therefore, although the speedup of your program is larger than that of your neighbor, it might well be that the execution time of your neighbor's program is less than yours. So, in fact we have to conclude that your neighbor's program is better, because it is faster! The same problem with speedup and efficiency arises when you start comparing parallel computers. Again, it might well be that the speedup on computer A is larger than that on computer B, for the same parallel program, but that computer B is in fact faster. This situation is easily accomplished by assuming that computer B has faster processors than computer A, but still has the same communications network. Just go back to the example of the vibrating string and look at the fractional communication overhead (Eq. 40). This number will be larger for computer B (why?) resulting in a smaller speedup for computer B. However, from Eq. [38] it is immediately clear that the execution time on computer B will be smaller than that on computer A. Again, do not compare speedups on different parallel computers.

Despite these subtle problems, speedup and efficiency are very useful metrics to evaluate the performance of a single parallel program. They tell you how well you did the job of parallelizing a program. A detailed speedup analysis will clearly show the overheads that e.g. result in a degrading performance if the number of processors is increased. Therefore, speedup and efficiency tell us how parallel programs behave, and how their performance can be improved. However, especially when you start comparing computers or parallel programs, speedup and efficiency should not be used, and you should go back to execution times. This fact will be further explored in section 4.4 on benchmarking.

4.2.2 Harmonic Mean Performance

Consider a parallel computer which executes m programs in various modes (sequential, parallel -, or vector mode). Each program has its own performance. Our task now is to define a *mean* performance for this specific program mix. This is a typical situation encountered with benchmarking suites, where the performance of a computer is measured by measuring the execution rates of a set of benchmark programs, and next calculating a mean performance. How can this be done?

Several methods can be used to calculate a mean performance. We will first consider arithmetic – and geometric mean performance and show why these metrics not are suited for our goal. Next we introduce the harmonic mean performance, which has all desired properties for a mean performance metric.

Assume that each program has an execution rate R_i, $1 \leq i \leq m$. R_i is e.g. measured in Mflop/s. The arithmetic mean performance rate R_a is defined as

$$R_a = \sum_{i=1}^{m} R_i / m . \qquad [44]$$

One can also define a weighted arithmetic mean rate as

$$R_a^* = \sum_{i=1}^{m} f_i R_i , \qquad [45]$$

where f_i are the weighting factors such that

$$\sum_{i=1}^{m} f_i = 1, \quad f_i \geq 0 . \qquad [46]$$

The problem with the arithmetic mean performance rate is that it is proportional to the sum of the inverses of the execution times of the *m* programs (because the execution rates R_i are inversely proportional to execution times). However, we would like to have a mean performance that is proportional to the inverse of the sum of the execution times. Therefore, the standard arithmetic mean performance rate is not suited.

Another possibility is the geometric mean performance rate, defined as

$$R_g = \prod_{i=1}^{m} R_i^{1/m} . \qquad [47]$$

Again, using the weighting factors of Eq. [46], one can also define a weighted geometric mean performance rate as

$$R_g^* = \prod_{i=1}^{m} R_i^{f_i} . \qquad [48]$$

The geometric arithmetic mean performance rate also does not have the desired property of being proportional to the inverse of the sum of the execution times of the *m* programs.

We need a mean performance based on arithmetic mean execution time. Realizing that $1/R_i$ is proportional to the execution time for program i, we first define the arithmetic mean execution time as

$$T_a = \frac{1}{m} \sum_{i=1}^{m} 1/R_i . \qquad [49]$$

Next, we define the harmonic mean performance rate as the inverse of the arithmetic mean execution time,

$$R_h = \frac{1}{T_a} = \frac{m}{\sum_{i=1}^{m} 1/R_i} . \qquad [50]$$

As before the weighted harmonic mean performance rate is defined as

$$R_h^* = \frac{1}{\sum_{i=1}^{m} f_i / R_i} . \qquad [50]$$

The harmonic mean performance rate has the desired property that it is proportional to the inverse of the sum of the execution times of the *m* programs. In that sense it is the most suitable metric to study mean performance of a program mix on a computer.

Consider, as an example, two programs with performance rates R_1 = 30 units/s and R_2 = 60 units/s. The task is to calculate a mean performance for this case. Before applying the formula of above, we also assume that each program has an amount of work of 60 units. This allows us to calculate the mean performance by first calculating the execution times, which are for both programs t_1 = 60/30 = 2 s and t_2 = 60/60 = 1 s. This results in a mean

performance of (60+60)/(2+1) = 40 units/s. Next, apply the formula for the mean performance as defined earlier:

$$R_a = \frac{30+60}{2} = 45 \text{ units/s},$$

$$R_g = 30^{1/2} \cdot 60^{1/2} = 42 \text{ units/},$$

$$R_h = \frac{2}{\frac{1}{30}+\frac{1}{60}} = 40 \text{ units/}.$$

Clearly, the harmonic mean performance produces the desired answer. We proceed to a second case by assuming that program 1 still has 60 units of work to do, but program 2 only has 30 units of work. In that case t_1 = 2 s and t_2 = ½ s, which results in a mean performance of (60 + 60)/(2 + ½) = 36 units/s. Now we must apply the formula for the weighted mean performances. The weight factors are determined by the fraction of the amount of work of each program, i.e. f_1 = 60 / (60 + 30) = 2/3 and f_2 = 30 /(60 + 30) = 1/3. Calculation of the mean performances now results in

$$R_a^* = \frac{2}{3} \cdot 30 + \frac{1}{3} \cdot 60 = 40 \text{ units/s},$$

$$R_g^* = 30^{2/3} \cdot 60^{1/3} = 38 \text{ units/s},$$

$$R_h^* = \frac{2}{\frac{2}{3 \cdot 30}+\frac{1}{3 \cdot 60}} = 36 \text{ units/s}.$$

Again, the harmonic mean performance results in the desired result. Finally, notice from these examples that

$$R_h < R_g < R_a . \qquad [51]$$

This property is true in general.

The concept of harmonic mean performance rates can also be applied to calculate harmonic mean speedup. Suppose we have a parallel program that, during execution, may use a different number of processors in different time periods. Instead of a mix of programs, we have one program that has a mix of usage of parallel processors (or, using terminology of the next section has different degrees of parallelism). We will apply the concept of harmonic mean performance to calculate a mean speedup for this program.

The program executes in mode i, if i processors are used. The performance rate R_i reflects the collective speed of i processors. Suppose that

$$T_1 = 1 \big/ R_1 = 1, \qquad [52]$$

is the sequential execution time on one processor. Suppose further that the program is executed in p modes with a weight factor f_i. A weighted harmonic mean speedup is defined as

$$S = \frac{T_1}{T^*} = \frac{1}{\sum_{i=1}^{m} f_i / R_i}, \qquad [53]$$

where T^* is the weighted arithmetic mean execution time across the p executing modes. As an example, assume that $T_i = 1/i$, or $R_i = i$. Clearly this is the ideal situation of linear speedup. Next, assume the following three sets of weight factors,

uniform distribution, $f_i = 1/p$ for all i

distribution which favors using $f_i = i/s$

more processors,

distribution which favors using fewer processors. $f_i = (p + 1 - i)/s$

Here, p is the total number of processors and i runs from 1 to p. The parameter s is defined as

$$s = \sum_{i=1}^{p} i .$$

Notice that all these workloads obey the constraints in Eq. [46]. Figure 41 shows the resulting harmonic speedups for these three workloads. As expected, the largest speedups are for those workloads that favor the use of more processors.

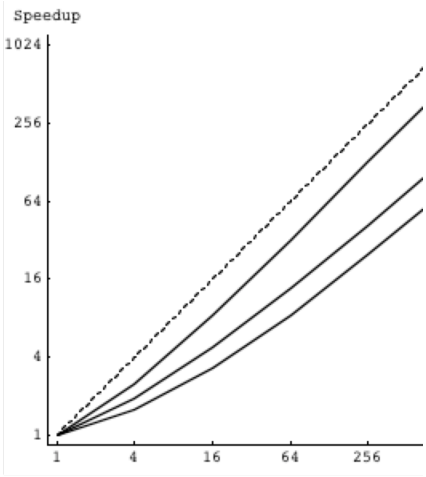


Figure 41: The harmonic mean speedup as a function of the number of processors. The dashed line is the ideal linear speedup curve. The upper solid line is for the second workload favoring the use of more processors, the middle solid curve is the first workload which has equal use of all numbers of processors, and finally the lower solid curve is for the third workload that favors use of fewer processors.

4.2.3 The Ruby Lee parameters

Besides speedup and efficiency, Ruby Lee defined some extra performance metrics. Although these metrics are not heavily used, we list them here for completeness. Define O_p as the total number of unit operations performed by a p-processor system. Furthermore, T_p is the execution time, measured in unit time steps. We assume that $T_1 = O_1$, and in general for $p > 1$ we have $T_p < O_p$. Ruby Lee defined the following metrics. First, the speedup factor is defined in its normal way,

$$S_p = \frac{T_1}{T_p} . \qquad [54]$$

The system efficiency is also defined in its normal way, as

$$\varepsilon_p = \frac{S_p}{p} = \frac{T_1}{pT_p} . \qquad [55]$$

The redundancy of a parallel computation is defined as

$$R_p = \frac{O_p}{O_1} \qquad [56]$$

and signifies the matching between software and hardware parallelism. The system utilization of a parallel computation is defined as

$$U_p = R_p \varepsilon_p = \frac{O_p}{pT_p}, \qquad [57]$$

and indicates the percentage of resources that was kept busy during the execution of a parallel program. Note the following relations between all these parameters:

$$\begin{aligned} &1 \le R(p) \le p, \\ &1/p \le E(p) \le U(p) \le 1, \\ &1 \le R(p) \le 1/E(p) \le p. \end{aligned} \qquad [58]$$

Try to prove these relations yourself. Finally, Lee defined the quality of a parallel computation as a parameter, which is proportional to the speedup and efficiency and inversely related to the redundancy. Thus we have

$$Q_p = \frac{S_p \varepsilon_p}{R_p} = \frac{T_1^3}{pT_p^2 O_p}. \qquad [59]$$

To summarize:

Speedup indicates the speed gain in a parallel computation;

Efficiency measures the useful portion of the total work performed by p processors;

Redundancy measures the extent of workload increase;

Utilization indicates the extent to which resources are utilized during a parallel computation;

Quality combines the effects of speedup, efficiency, and redundancy into a single expression to assess the relative merit of a parallel computation on a computer system.

4.3 Time Complexity Models

4.3.1 Introduction

We will now study performance and scalability from a purely theoretical point of view. By considering hypothetical "ideal" parallel computers and synthetic workloads we are already able to draw some very important conclusions with respect to performance and scalability. We will encounter the Amdahl bottleneck and solutions to it, and finally end up with the scaled speedup models of Sun and Ni.

4.3.2 Average parallelism and Obtained Speedup

Consider a hypothetical parallel computer with an infinite number of equal processors and negligible communication latency between the processors. Furthermore, any overheads due to I/O can be neglected. This parallel computer runs in MIMD mode, but because of the zero communication latency, the clocks of all processors can be synchronized. This allows us to establish how many processors are used during the execution of a parallel program at a certain time period.

Define the *Degree of Parallelism* (DOP) as the number of processors in use to execute a parallel program during a certain period of time. The DOP as a function of time is called the *parallelism profile* of a program. As an example, consider the parallel addition of 16 numbers on the ideal computer. As is shown in Figure 42, this is established by pair-wise

additions in a tree structure. After 4 time slices the addition is completed. The corresponding parallelism profile is drawn in Figure 43.

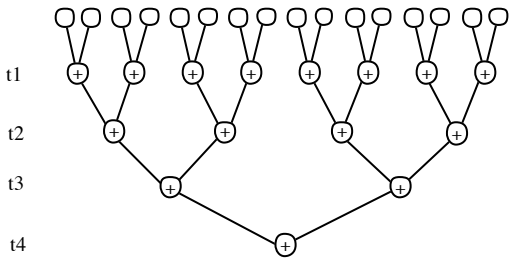


Figure 42: parallel addition of 16 numbers by pair-wise addition

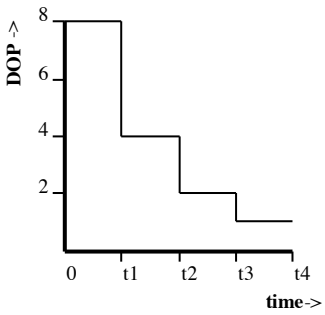


Figure 43: Parallelism profile of the parallel addition of 16 numbers

The *Average Parallelism A* is computed as the average DOP in a parallelism profile:

$$A = \frac{1}{T}\int_0^T \mathrm{DOP}(t)dt\,, \qquad [60]$$

with T the total run time of the parallel program on the ideal parallel computer. In discrete form we can write

$$A = \sum_{i=1}^{m} i \cdot t_i \Big/ \sum_{i=1}^{m} t_i\,, \qquad [61]$$

where t_i is the amount of time during which DOP = i, and m is the maximum DOP in the parallelism profile. The denominator of Eq. [61] can be identified with the run time of the parallel program on p processors and the numerator with the execution time on 1 processor. Therefore, the average parallelism A is equal to the speedup on the ideal parallel computer, and is an upper bound to the speedup that can be reached on real machines. For the addition example we find that $A = (8+4+2+1)/(1+1+1+1) = 15/4$.

Define Δ as the computing capacity of a processor (expressed in e.g. Mflop/s). The amount of work executed while running a part of the program with DOP = i is

$$W_i = \Delta i t_i\,, \qquad [62]$$

and the total amount of work is

$$W = \sum_{i=1}^{m} W_i\,. \qquad [63]$$

Now assume that the workload W is executed on p processors. The execution time for the portion of work with DOP = i is

$$t_i(p) = \frac{W_i}{i\Delta}\left\lceil \frac{i}{p} \right\rceil. \qquad [64]$$

Notice that in the formulation of Eq. [64], possible load imbalance is implicitly taken into account. The total execution time of workload W on p processors, $T_p(W)$, equals

$$T_p(W) = \sum_{i=1}^{m} t_i(p) = \sum_{i=1}^{m} \frac{W_i}{i\Delta} \left\lceil \frac{i}{p} \right\rceil . \qquad [65]$$

Next we define the (communication) overhead for a p processor system for completion of the workload W as $Q_p(W)$, and put $Q_1(W) = 0$. We can now define the *Obtained Speedup* for the workload W on a p processor system as

$$S_p(W) = \frac{T_1(W)}{T_p(W) + Q_p(W)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i} \left\lceil \frac{i}{p} \right\rceil + \Delta Q_p(W)} . \qquad [66]$$

Notice that the obtained speedup as defined in Eq. [66] is equal to the earlier defined relative speedup. In the sequel we will use the obtained speedup to analyze scaled speedup laws and scalability.

The *Asymptotic Speedup* is found by taking the limit of an infinite number of processors in the obtained speedup, i.e.

$$S_\infty = \lim_{p \to \infty} S_p(W) = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i} + \Delta Q_m(W)} . \qquad [67]$$

Note that in Eq. [67] the (communication) overheads have settled at $Q_m(W)$ and that $S_\infty = S_m$. If all overheads Q are ignored, we immediately find that

$$S_\infty(W) = A . \qquad [68]$$

Try to prove this relation yourself. In general we have

$$S_\infty(W) \le A . \qquad [69]$$

This equation is very important, because it tells us that the average parallelism A is the upper bound to speedup. If you know A, you have the best speedup you will ever get. In fact, it is interesting to compare A with the real obtained speedup S_p. The difference between the two shows the influence of all kinds of overheads. Therefore, several tools to extract A from real parallel programs have been developed.

4.3.3 Amdahl's Law

In 1967 a computer scientist at IBM, Dr. G. Amdahl, formulated a speedup law that came as a shock for those people working on vector computing and parallel computing [39]. Amdahl's arguments seemed to show that massively parallel computing is out of the question, because of the so-called sequential bottleneck. As you are currently in the middle of a lecture on parallel computing, you may already suspect that there is a way out of Amdahl's arguments, and that is indeed true. The escape route from Amdahl's sequential bottleneck leads us to the very important concept of scaled workloads and scalability. In this section we will first study Amdahl's law in some detail.

Assume that a parallel program consists of two workloads, a completely sequential - and a completely parallel part. Furthermore assume that the sequential workload is a fraction α of the complete workload. Finally, ignore all overheads. The execution time of this workload on a p processor system is

$$T_p = \alpha T_1 + \frac{1-\alpha}{p} T_1 . \qquad [70]$$

The resulting obtained speedup is

$$S_p = \frac{1}{\alpha + (1-\alpha)/p}, \qquad [71]$$

and the asymptotic speedup becomes

$$S_\infty = \frac{1}{\alpha}. \qquad [72]$$

Equation [71] is known as Amdahl's law. Before we discuss the implications of Amdahl's law we will first derive the average parallelism for the Amdahl workload and show that it is equal to the asymptotic speedup of Eq. [72]. The approach will be to first assume that the maximum DOP is p, then draw the parallelism profile and calculate A, and finally let p go to infinity. The parallelism profile is drawn in Figure 44, where we assume that the parallel part is running on p processors of the ideal machine. Of course, execution of the sequential and parallel part may be interleaved, but in Figure 44 both parts are lumped together to clarify the picture. We assume that $T_1 = 1$. In that case the time spent in the parallel part is $(1-\alpha)/p$ and in the sequential part is α. Note that the parallelism profile in Figure 44 is not drawn to scale.

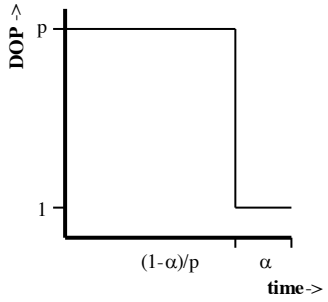


Figure 44: The parallelism profile for the Amdahl workload.

For this parallelism profile we find

$$A = \frac{p \times (1-\alpha)/p + 1 \times \alpha}{(1-\alpha)/p + \alpha} = \frac{1}{\alpha + (1-\alpha)/p}. \qquad [73]$$

which actually is equal to Amdahl's law, Eq. [71]. In Amdahl's case the parallelism is not limited to p but is in fact infinite. Therefore, the avarage parallelism is Amdahl's case is $A = 1/\alpha$, which is indeed equal to the asymptotic speedup in Eq. [72].

What then is so shocking about Amdahl's law? The point is that the maximum speedup is limited by the sequential part of the program. What would be a reasonable assumption for the sequential fraction of a program? Suppose it is 1 %. In that case the asymptotic speedup is as small as 100. This means that massive parallelism, where we want to use in an efficient way thousands of processors, just does not seem to be possible. Amdahl's argument raised many questions regarding the possibility of massive parallel computing. The argument becomes even worse if we include the overheads. What to do next? You know that parallel computers with thousands of processors are in operation, so, where is the answer? How can we escape from Amdahl's $1/\alpha$ sequential bottleneck? Just think a while about this question before moving to the next section.

4.3.4 Gustafson's Answer

Amdahl's law raised a lot of skepticism about the possibility of getting to very high speedups. Another computer scientist, Gordon Bell, challenged the research community with a bet. He said that no one could demonstrate a speedup of more than 1000 on a real machine for a real parallel application, and those who actually would be able to do this,

would gain $1000,-. In 1988 it was pay day for Gordon Bell, because Gustafson and Bariss, from Sandia National Laboratories in the USA, actually demonstrated a speedup of 1016 to 1021 on a 1024 processor parallel computer [40]. Gustafson's procedure and its formalization in a law is now known as Gustafson's speedup law.

In Amdahl's law the amount of computational work is kept constant as the number of processors is increased. This so-called *fixed-load* leads to Amdahl's sequential bottleneck and prevents to reach very high speedups. Although the fixed-load constraint is essential in some application areas (e.g. real time control applications, embedded software) the situation is very different in many engineering and research applications. Usually what happens if a researcher or engineer gets a faster computer is that he will not run his current problems faster. No, he will take a much larger problem, and run it on the faster computer in approximately the same amount of time. This truly is a typical situation. A researcher studying e.g. galaxy formation (using e.g. the N-body problem of section 2.5, but now with a very large number of stars) may be perfectly happy with execution times of a week. If he gets a faster computer he will not run faster, but immediately increase the number of stars in his simulation, so that he can get more detailed results. The same is true for an engineer designing cars. He might run crash simulations in order to increase the safety of a car design. In this case execution times of a few hours are acceptable. Again, with a faster computer, the engineer will probably increase the accuracy of the models (finer grid spacing, more detailed physics in the model) and run his more accurate model on the faster computer in again a few hours of execution time.

Gustafson realized this, and formulated the *fixed-time* concept, which results in scaled speedup models. Assume, if the problem size is increased, that the sequential work of the program remains constant and that the parallel work grows. In a first approximation this behavior is observed in many applications. The serial part consists of I/O, initialization, and the like, which are independent of problem size. Now assume that the problem size is increased in such a way that the total execution time of the parallel part, running on p processors, also remains constant. In doing so, we scaled the workload in such a way that the execution time remains constant, i.e. a fixed-time scaling. What Gustafson did was to calculate the speedup, not of the original workload, but of the scaled workload.

We will now derive Gustafson's fixed-time speedup law. As in Amdahl's law we again assume that we have a sequential and a fully parallel workload, and that the sequential fraction of the original workload equals α. The upper case quotes in the following equations refer to the scaled problem. First, we have, by definition,

$$\begin{aligned} T_1 &= T_{seq} + T_{par}\,, \\ T_1' &= T'_{seq} + T'_{par}\,, \\ T'_p &= T'_{seq} + T'_{par} / p\,. \end{aligned} \qquad [74]$$

Next we apply the fixed-time assumption, $T'_p = T_1$ and use the assumption that $T'_{seq} = T_{seq}$, which, when inserted in Eqs. [74], finally results in

$$T_{par} = T'_{par} / p\,. \qquad [75]$$

Next we calculate the obtained speedup for the scaled problem, i.e.

$$S'_p = \frac{T_1'}{T'_p} = \frac{T_{seq} + pT_{par}}{T_{seq} + T_{par}} = \alpha + p(1-\alpha)\,. \qquad [76]$$

The final expression in Eq. [76] is Gustafson's law. Clearly, the scaled speedup does not suffer from the sequential bottleneck. Before taking a closer look at Gustafson's law we first derive once more from the average parallelism. We follow the same approach as in Amdahl's law. Figure 45 shows the parallelism profile for the fixed-time situation. Notice that the time spent in the parallel part, i.e. with DOP = p is now 1-α.

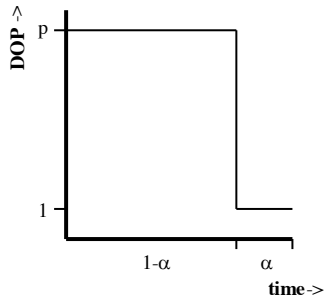


Figure 45: The parallelism profile for the fixed-time situation (Gustafson)

We immediately find for the average parallelism, which in this case is again equal to Gustafson's obtained speedup on an ideal machine,

$$A = \frac{p(1-\alpha)+\alpha}{(1-\alpha)+\alpha} = \alpha + p(1-\alpha). \qquad [77]$$

By proportional scaling of the parallel workload with the number of processors, the speedup does not suffer from the sequential bottleneck, and large speedups are possible. In Figure 46 the obtained speedup for the fixed-load constraint and the fixed-time constraint are drawn for p = 1024. We can conclude that in the fixed-time constraint situation large speedups are possible.

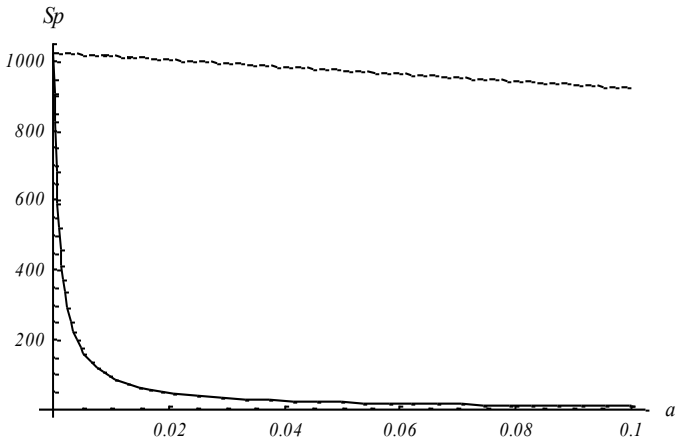


Figure 46: The obtained speedup on 1024 processors, as a function of the fraction of sequential work α. The solid line is the fixed-load Amdahl case, the dashed line is the fixed-time Gustafson case.

It is clear that scaling the workload is the answer to circumventing Amdahl's sequential bottleneck. In fact, what we did was to increase the problem size by increasing the parallel workload and keeping the sequential workload equal. In other words, we have in fact decreased the sequential fraction of the scaled problem. Let us try to calculate this scaled fraction. By definition

$$\alpha' = \frac{T'_{seq}}{T'_{seq} + T'_{par}} = \frac{T_{seq}}{T_{seq} + pT_{par}}. \qquad [78]$$

In the second step we applied the fixed time assumptions, leading to Eq. [75]. After some algebra we find

$$\alpha' = \frac{\alpha}{\alpha + p(1-\alpha)}. \qquad [79]$$

For the scaled problem we can now calculate the obtained speedup using Amdahl's law, i.e. apply Eq. [71],

$$S'_p = \frac{1}{\alpha' + (1-\alpha')/p}. \qquad [80]$$

By substituting Eq. [79] in Eq. [80] we again recover the Gustafson fixed-time speedup of Eq. [76].

4.3.5 Scalable Computing

Scalable computing, i.e. increasing the workload if the number of processors of a parallel computer is increased, is a prerequisite to obtain high speedups. The topic of *scalability* therefore is important. We define scalability as the study of how the performance of a parallel program behaves as a function of the number of processors and as a function of the problem size. Furthermore, scalability tells us how the problem size should be scaled up if the number of processors is increased in order to maintain a certain performance level. Figure 47 shows several scaling strategies and resulting efficiencies. We cannot scale at all, i.e. Amdahl's fixed work concept or we can apply different forms of linear scaling, or adapt other, e.g. exponential, workload scaling strategies. In this section we will study such scaling strategies, and derive formulas for scaled speedup laws.

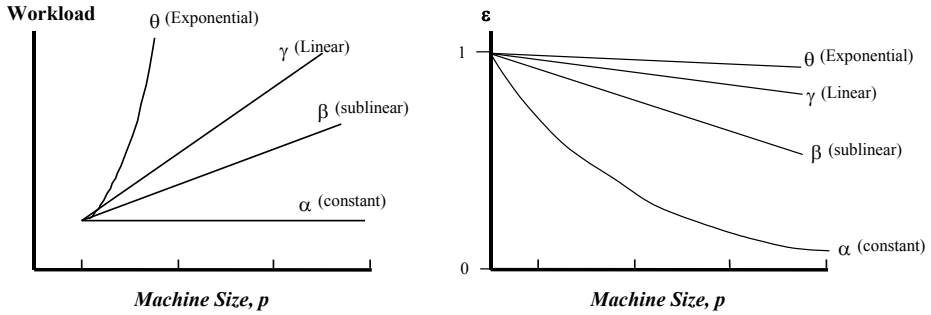


Figure 47: Scaling the workload with the machine size on the left, and resulting efficiencies on the right.

When scaling the workload on real parallel computers we will run into a number of constraints. First, memory is finite. Although we assume, if the number of processors is increased, that the total memory in the parallel computers increases as well, memory does present an upper bound on the scaling of workload. Furthermore, another bound is presented by communication. If the amount of communication becomes so large that it completely overwhelms computation, the efficiency drops dramatically. Typically, memory bounds are encountered for too large problem sizes, and communication bounds for too many processors. This is drawn schematically in Figure 48. Furthermore, this figure also shows the three scaling models that we will study in more detail. You already encountered the fixed-load and fixed-time models. As a third scaling model we will also consider the fixed-memory model.

The fixed-load model assumes a constant workload (i.e. Amdahl's case). Here, scaling will be limited by the communication bound. In the fixed-time model (i.e. Gustafson's case) the execution time of the parallel program is kept constant. A final scaling model that we will discuss is due to Sun and Ni and is the fixed-memory model. Here the problem is scaled such that all memory is consumed, and is clearly limited by a memory bound.

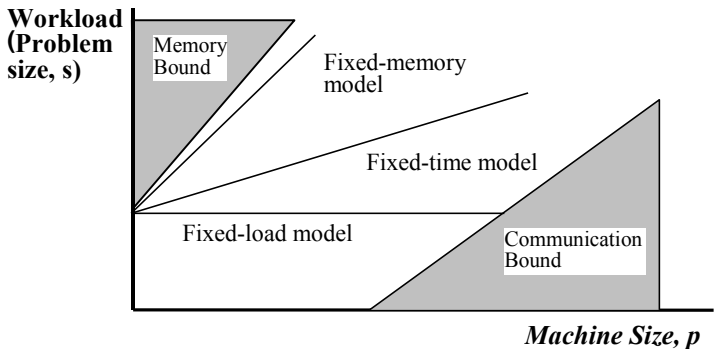


Figure 48: Scaling the workload in the presence of memory and communication bounds.

Remember the equation that we derived for the obtained speedup, i.e. Eq. [66]. In terms of scalable computing we call this the fixed-load speedup factor. Eq. [66] is a general expression for the fixed-load speedup factor. If we assume that

$$\begin{aligned} &Q_p(W) = 0, \\ &W_i = 0 \text{ if } i \neq 1 \text{ or } i \neq p, \end{aligned} \qquad [81]$$

i.e. the workload only contains a sequential part W_1 and a parallel part W_p, and the communication overheads are zero, and substitute these expressions into Eq. [66] we find

$$S_p = \frac{W_1 + W_p}{W_1 + W_p / p}, \qquad [82]$$

which is, as it should be, Amdahl's law (why?).

Next consider scaling the workload. We start again with the fixed time scaling, and then move to the more general case. Assume once more a workload W and a scaled workload W'. Let m' be the maximum DOP of the scaled workload. In general

$$W_i' > W_i \text{ for } 2 \le i \le m \text{ and } W_1' = W_1. \qquad [83]$$

In the fixed-time speedup model we assume

$$T_1(W) = T_p(W') + \text{overheads}, \qquad [84]$$

or

$$\sum_{i=1}^{m} W_i = \sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{i}{p} \right\rceil + \Delta Q_p(W'). \qquad [85]$$

We now obtain the fixed-time speedup as

$$S_p' = \frac{T_1(W')}{T_p(W')} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{1}{p} \right\rceil + \Delta Q_p(W')} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i}. \qquad [86]$$

Eq. [86] is a general formula for the fixed-time speedup factor. The scaled workload W' should adhere to the fixed time constraint in Eq. [85]. By again imposing the special workload distribution of Eq. [81] we obtain Gustafson's law. By substituting Eq. [81] into Eq. [85] we find

$$W'_p = pW_p, \qquad [87]$$

and combining this with Eq. [86] results in

$$S'_p = \frac{W_1 + pW_p}{W_1 + W_p} \ . \qquad [88]$$

In this way you recover Gustafson's law.

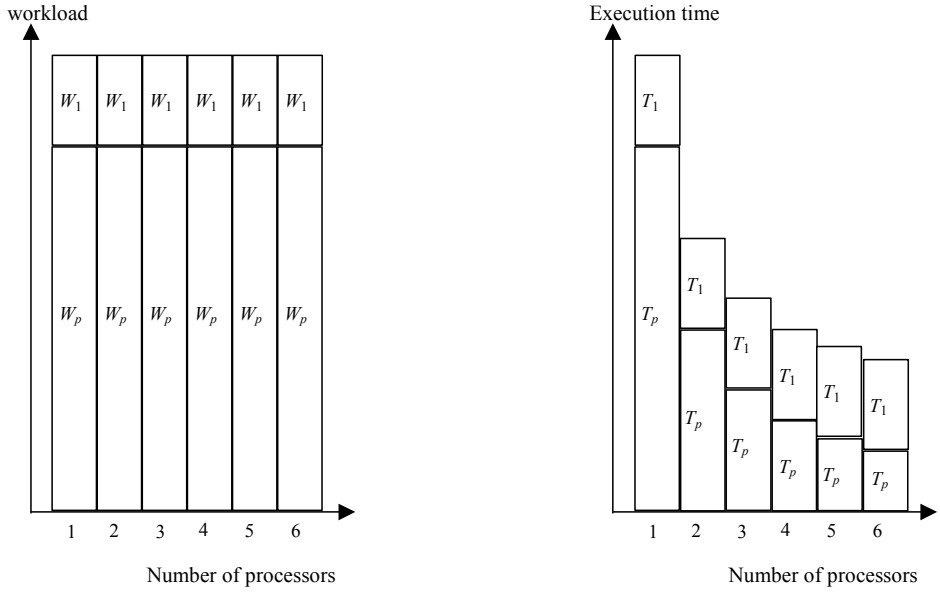


Figure 49: Scaling of workload and execution time in case of Amdahl's fixed-load.

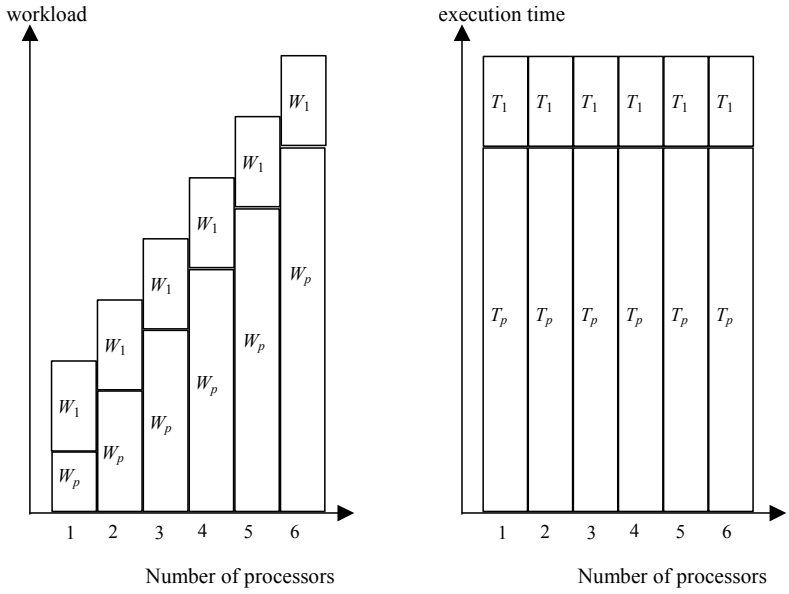


Figure 50: Scaling of workload and execution time in case of Gustafson's fixed-time scaling..

It is instructive to visualize this scaling of workload, and the resulting execution times, for the Amdahl and Gustafson case. This is shown for Amdahl's case in Figure 49 and for Gustafson's case in Figure 50.

We will now draw our attention to the final scaled speedup model, which can be viewed as a generalization of Amdahl's and Gustafson's law, and, as will become clear, results in a very general expression for scaled speedup. This is the fixed-memory speedup model, due to Sun and Ni [41]. The idea is maximize the use of both CPU and memory. So, instead of keeping the execution time fixed, the problem size is scaled such that all memory is used. This will usually result in a further increase in execution time as compared to the original problem. In other words, if you have adequate memory space and the scaled problem meets the time limit imposed by Gustafson's law, you can further increase the problem size, yielding an even better or more accurate solution. Define the scaled workload as W^*, then the fixed-memory speedup is defined as

$$S_p^* = \frac{\sum_{i=1}^{m^*} W_i^*}{\sum_{i=1}^{m^*} \frac{W_i^*}{i} \left\lceil \frac{i}{p} \right\rceil + \Delta Q_p(W^*)}. \qquad [89]$$

To proceed we first need to derive expressions for the relation between the scaled and original workload, by considering the relation between the work and related memory usage. In the sequel we will restrict ourselves to the special workload of Eq [81] and also again neglect overheads. Under those conditions the memory bounded speedup becomes

$$S_p^* = \frac{W_1^* + W_p^*}{W_1^* + W_p^* / p}. \qquad [90]$$

The workload of the sequential part is assumed to be independent of both problem size and system size:

$$W_1^* = W_1. \qquad [91]$$

The scaled parallel workload needs some more consideration. The workload W and the memory requirement M for this workload are related by $W = g(M)$. Assume that $g(x)$ is a semihomomorphism[1] and that the total memory capacity M of one processor is available for the workload W_p, i.e.

$$W_p = g(M). \qquad [92]$$

On p processors the memory increases with a factor p to pM. On an ideal parallel system, without communication latencies, remote data access is equivalent to local data access and therefore replication of data is not necessary. In fact, for any Shared Address space system replication of data is not strictly necessary. The total available memory can now be used for the parallel portion of the scaled problem, i.e.

$$W_p^* = g(M^*) = g(pM) = G(p)g(M) = G(p)W_p$$

The function $G(p)$ describes the increase in parallel workload after increasing the total amount of memory in the system with a factor p. The resulting fixed-memory speedup factor is

$$S_p^* = \frac{W_1 + G(p)W_p}{W_1 + G(p)W_p / p}. \qquad [93]$$

Let us investigate three special cases.

---

[1] A function $g(x)$ is a semihomomorphism if $g(cx) = G(c)g(x)$. For instance, the function $g(x) = ax^b$ is a semihomomorphism with $G(x) = x^b$.

1. $G(p) = 1$. This corresponds to the fixed-problem size and equation 14 reduces to Amdahl's law.
2. $G(p) = p$. The workload increases linearly with the available memory, keeping the total execution time fixed. This corresponds to Gustafson's law.
3. $G(p) > p$. Here, the workload increases faster than the memory requirements of the parallel program, and the resulting speedup is larger than the fixed-time scaled speedup.

In order to find expressions for $G(p)$ we usually have to perform an order of magnitude analysis, where we only keep the highest order terms. We will investigate an example of a parallel matrix-matrix product

The matrix-matrix product **AB** = **C** requires to store three n×n matrices, therefore the memory requirement is $M = 3n^2$. The total work (assuming that it can be done in parallel) is $W_p = n^2(2n - 1) \sim 2n^3$ for large n. Therefore $W_p = 3^{-3/2} \times 2\, M^{3/2}$, and we immediately find $G(p) = x^{3/2}$ and

$$W_p^* = p^{3/2} W_p \,. \qquad [94]$$

This is an example where $G(p) > p$, and the scaled speedup is even better than for fixed-time speedup. The fixed-memory speedup for this example is

$$S_p^* = \frac{W_1 + p^{3/2} W_p}{W_1 + p^{1/2} W_p} \,. \qquad [95]$$

4.3.6 Scalability

Scalability determines the degree of matching between a computer architecture and an application algorithm. For different *Computing Systems* (i.e. architecture - algorithm pairs) the analysis should result in statements about the efficiency of the computing system, e.g. in terms of number of processors in relation to workload.

Scalability can be affected by

- machine size - number of processors;
- Clock rate;
- Problem size - the amount of computational work is determined by the problem size;
- CPU time - The actual execution time of a given program on a given p-processor system;
- I/O demands;
- Memory capacity;
- Communication overhead - Amount of time spent in inter-processor communication, synchronization, etc.;
- Computer Cost;
- Programming overhead.

Depending on the computational objectives and resource constraints, one will fix some of the parameters and optimize the remaining ones. Scalability must be able to express the effects of the architecture's interconnection network, of the communication patterns inherent to the algorithm, of the physical constraints imposed by technology, and of the cost effectiveness or system efficiency.

One approach to scalability is the *isoefficiency* concept. Here, one studies how one should increase the workload of a parallel algorithm, running on a parallel computer, as one increases the number of processors, such that the efficiency maintains at a fixed value. Kumar and Rao introduced the isoefficiency concept in 1987.

To proceed we have to write the efficiency in terms of the total workload $W(s)$ of an algorithm as a function of the problem size s, and a function $h(s,p)$ which is the lump sum of all overheads as a function of s and p, the number of processors. From previous discussion we know that we can write

$$T_1 = \frac{W(s)}{\Delta} \qquad [96]$$

and

$$T_p = \frac{1}{p}\frac{W(s)+h(s,p)}{\Delta}. \qquad [97]$$

Can you write down the relations between the overhead function h and the fractional overheads or the overhead function Q from previous sections? With these definitions we find for the efficiency

$$\varepsilon = \frac{W(s)}{W(s)+h(s,p)}. \qquad [98]$$

Rewrite this equation to

$$W(s) = \frac{\varepsilon}{1-\varepsilon}h(s,p). \qquad [99]$$

Eq. [99] tells us how the workload should grow in proportion to the overhead in order to maintain a fixed efficiency ε. Therefore the *isoefficiency* function is defined as follows

$$f_E(p) = \frac{\varepsilon}{1-\varepsilon}h(s,p).$$

If the workload grows as fast as the isoefficiency function, a constant efficiency can be maintained for a given algorithm-architecture combination.

4.4 Benchmarking

4.4.1 Introduction

The goal of benchmarking is to measure, in a controllable way, the performance of a computer. In itself this already is a difficult task, and if the computer in question is parallel, benchmarking becomes even more difficult. R. Hockney has written a very nice introduction into the field of computer benchmarking [42]. The idea of benchmarking is to define a standard set of measurements (benchmark programs) to assess the performance of computing systems. Reasons to do so are (quoting from Hockney, Ref. [42]):

- To establish and maintain high standards of honesty and integrity (see 12 way to fool the masses...),
- To improve the status of performance analysis as a rigorous scientific discipline,
- To reduce confusion in the HPC field,
- To increase understanding of HPC systems, both low-level HW and SW and high-level system performance,
- To assist the purchaser of HPC equipment,
- To reduce the amount of time needed for benchmarking,
- To provide feedback to vendors on bottlenecks that can be alleviated in future products.

Like all experimental disciplines, the implementation and execution of *controllable* experiments, which allow for *reproducibility* of the results by other groups, is of crucial

importance for benchmarking. A specialized Benchmark organization exists (see Ref. [43) and several web sites publish benchmark results, see e.g. [44].

We will first consider a few performance measures and well-known benchmarks. Next, we will discuss fundamental metrics that were introduced by Hockney, and finally discuss an example of a benchmark suite for parallel computers, the Parkbench.

4.4.2 Performance Measures

Performance of a computer can be measured in several ways. First, one can simply quote the number of instructions that a computer can execute per second. This is usually expressed as Million Instructions per Second, or MIPS rating. One can also look at floating point performance by measuring the number of floating point operations that are issued per second. This is usually expressed in Million floating-point operations per second (Mflop/s, see also section 2.1.1). On first sight these two performance measures seem all you need. Unfortunately, they depend heavily on the specific application that is used to measure the performance. Therefore, people started to define benchmark suites, i.e. a set of programs that together cover a broad spectrum of possible instruction mixes. By measuring the performance of all programs and reporting statistics (such as harmonic mean performance, together with minimum and maximum performance), such benchmark suites give a reasonable idea of the performance of a computer.

The *Dhrystone* suite is a CPU intensive benchmark, consisting of a mix of about 100 high-level language instructions and data types found in system programming applications, where floating point operations are not used. The Dhrystone rating should be a measure of the integer performance of modern processors. The Dhrystone results are usually reported in units *kDhrystone/s*, but more often relative to the VAX11/780 processor. For instance, in version 2.1 of the benchmark, the VAX11/780 scored 1.6 KDhrystones/s, and results for other processors are reported relative to this, in units *Dhrystone MIPS_2.1*. The Dhrystone is discredited due to compiler recognition of the benchmark. Still, this test remains very popular in the area of minicomputers. Some recent results are reported in Table 6. A related benchmark suite is Whestone, a Fortran based synthetic benchmark, including both integer and floating point operations. Whestone does not contain any vectorisable code. Both Whetstone and Dhrystone are criticized for being unable to predict the performance of user programs. The sensitivity to compilers is a major drawback of these benchmarks. Only application oriented benchmarks will do the trick.

System	Dhrystone MIPS 2.1
DEC 21164 300 MHz	464.8
Intel Pentium Pro 200 MHz	446.9
MIPS R10000 195 MHz	214
PPC 604 132 MHz	196.7
UltraSPARC 167 MHz	179.0
PA-RISC7100 99.0 MHz	146.5

Table 6: Some Dhrystone results for a number of processors.

Before moving to benchmarks that are more suited for benchmarking for typical applications from engineering or scientific computing, we first mention two benchmarks that are related to single application domains. The first is the TPS rating, and is related to on-line transaction processing. The throughput of computers is measured in Transactions per Second, or TPS. Another is the KLIPS rating, which is related to Artificial Intelligence applications. In order to indicate the reasoning power of an AI machine, the Kilo Logic Interference per Second (KLIPS) rating is used.

Let us move to some benchmarks that are used to measure the performance of a single processor for more numerically oriented applications. The first is the LinPack benchmark. Here, a dense set of $n \times n$ equations is solved using Gaussian elimination. The performance

is measured in Mflop/s. In the original benchmark the dimension of the system of equations was $n = 100$. With the growth of computational power one usually also reports the performance for $n = 1000$. In a third version, called the LinPeak benchmark, one is allowed to choose any n, such that the performance is maximized. In its parallel form this benchmark is used to produce the top 500 results (see section 2.1.6). Table 7 shows some results for LinPack benchmarks. Real high performance is only reached for large system sizes. In this way the processor is able to keep its pipelines filled (think of the vector paradigm of parallelism) and run at very high speeds. Another important factor is the size of the cache. If the cache is large enough and the compiler smart enough, cache misses and memory bottlenecks can be avoided, again increasing the performance of the processor.

System	***n* = 100**	***n* = 1000**
HP i2000 Itanium 800 MHz	580	2282
AMD Athlon Thunderbird 1200 Mhz	558	
IBM RS/6000 450 MHz	503	1451
Hewlett-Packard V2600(550 MHz)	465	1221
AMD Athlon (600 Mhz)	260	557
SGI Origin 2000 (300 Mhz)	173	553
Sun UltraSPARC II, 336MHz	154	461
Pentium Pro 200 Mhz	38	
Macintosh 7300, PPC-604 200MHz	32	

Table 7: Some LinPack results in Mflop/s.

Measure	**Mflop/s**
Maximum Rate	43.1
Minimum Rate	4.9
Arithmetic Mean	16.8
Geometric Mean	14.2
Harmonic Mean	12.1

Table 8: Results for double precision Livermore loops for the PowerPC-601 at 80 MHz.

System	**Geometric Mean**
R12000 300Mhz	168
Pentium 3 450 MHz	72
DEC Alpha, 133 MHz	46.6

Table 9: The geometric, in Mflop/s, of the Livermore loops for some systems.

Another well-known single processor benchmark for numerical applications are the Livermore Loops. These are a set of 24 Fortran DO loops extracted from operational codes used at the Lawrence Livermore National Laboratory in the USA. They have been used since the early seventies to assess the arithmetic performance of computers and their compilers. They are a mixture of vectorisable and non-vectorisable loops and test rather fully the computational capabilities of the hardware and the skill of the software in vectorising and compiling code. The main value of the benchmark is the range of performance that it demonstrates. The benchmark provides the individual performances of each loop, together with various averages (arithmetic, geometric, harmonic). However, it is difficult to give a clear meaning to these averages, and the value of the benchmark is more in the distribution itself. In particular, the maximum and minimum give the range of likely performance in full applications. Table 8 and Table 9 give some results of the Livermore loops. Notice the enormous range in performance. Although processors may

run at a very high speed for certain applications, their performance is very poor for others. This strong dependence of performance on the specific application should never be forgotten.

4.4.3 Fundamental Metrics

In order to give benchmarking a strong base as an experimental science, Hockney defined a number of fundamental metrics. We already encountered some ways to measure performance, but did not define exactly how to measure it. Therefore, we need to stop for a while, and reconsider the fundamental metrics that we should use in performance measurement. In defining these metrics we assume that we are mainly interested in floating point performance of a parallel computer.

The most fundamental metric in benchmarking is *time*. We always measure wall clock time (and *not* e.g. CPU time of a process). When measuring time one should always specify the problem size N and the number of processors p, i.e. always measure $T(N, p)$. A second fundamental metric is the number of floating point operations that are executed by a program. For a correct counting one needs to agree on the number of operations for e.g. a division, or to compute e.g. an exponential. An agreed upon count is reproduced in Table 10. We are now able to actually count, in a well-defined way, the number of floating-point operations in a program. Define $F_B(N)$ as the number of floating-point operations in the sequential benchmark program, and $F_H(N, p)$ as the actual number of floating point operations which are performed by the parallel benchmark program on p processors. These definitions allow to capture the problems related with e.g. the differences between fair and relative speedup, and also capture the essence of the $O(p)$ parameter of Lee, see section 4.2.3.

Operation	**flop**
Add, subtract, multiply	1
Divide, square-root	4
Exponential, sine, etc.	8
IF(x.REL.y)	1

Table 10: Floating point operation (flop) count for a number functions.

We are now ready to define the following performance metrics.

1. *Temporal Performance*

If we want to compare the performance of different algorithms to solve the same problem, the correct metric to use is the Temporal Performance, defined as

$$R_T(N; p) = \frac{1}{T(N; p)}. \qquad [100]$$

In fact the temporal performance tells us how many problems we can solve per second.

2. *Simulation Performance*

This is a special case of Temporal Performance, which occurs for simulation programs in which the benchmark problem is defined as the simulation of a certain period of physical time. In this case we speak of *Simulation Performance* and use units such as simulated *days per day* in e.g. weather forecasting, or *simulated picoseconds per second* in electronic device simulations.

3. *Benchmark Performance*

In order to compare the performance of a computer on one benchmark with its performance on another, account must be taken of the different amounts of work

(measured in flop) that the different problems require for their solution. Using the flop count for the benchmark we can define the *Benchmark Performance* as

$$R_B(N;p) = \frac{F_B(N)}{T(N;p)}. \qquad [101]$$

The units of benchmark performance are Mflop/s (benchmark name), where we include the name of the benchmark in parentheses to emphasize that the performance may depend strongly on the problem being solved.

4. *Hardware Performance*

If we wish to compute the observed performance with the theoretical capabilities of the computer hardware, we must compute the actual number of floating point operations performed and from that derive the *Hardware Performance*, defined as

$$R_H(N;p) = \frac{F_H(N;p)}{T(N;p)}. \qquad [102]$$

We have now clearly defined what we mean with performance. Unfortunately, the performance depends always on N and sometimes also on p. It would be helpful if we could somehow reduce the number of free parameters. It turns out that, if the execution time depends linearly on N, it is possible to define two parameters that completely capture the N-dependence and that have a very nice interpretation. These are Hockney's (r_∞,$n_{½}$) parameters. This metric was originally used to analyze vector operations. It is also very useful for point-to-point communication, because here the communication time is also linear in the amount of data that is sent. Assume that we have a vector operation, i.e. a DO loop, of length N. We can assume a linear relationship between the execution time of the loop and the vector length, i.e.

$$T(N) = aN + b. \qquad [103]$$

If the number of operations per vector element is q, we can write for the benchmark performance of the loop

$$r(N) = \frac{qN}{aN + b}. \qquad [104]$$

The asymptotic performance rate is found by taking the limit of infinite N,

$$r_\infty = \lim_{N\to\infty} r(N) = \frac{q}{a}. \qquad [105]$$

The length of the vector at which the benchmark performance is half of the asymptotic performance rate is called the half performance length $n_{½}$. By definition $r(n_{½}) = ½\, r_\infty$, which results in

$$\frac{qn_{1/2}}{an_{1/2} + b} = \frac{q}{2a} \Rightarrow n_{1/2} = \frac{b}{a}. \qquad [106]$$

With these definitions we can write for the execution time and benchmark performance rate of the DO loop

$$t = q\left(n + n_{1/2}\right)/r_\infty\,,$$
$$r = \frac{r_\infty}{\left(1 + N/n_{1/2}\right)}. \qquad [107]$$

As an example consider the vector update operation (**y** := *c***x** + **y**). For each vector element we have to perform a multiplication and an addition, i.e. $q = 2$. Suppose that, after a careful experiment we found for the parameters of Eq. [103]: $a = 20$ ns and $b = 2$ μs. How would you set up such an experiment to measure the parameters a and b? From this experiment we find for the vector update operation on this specific computer

$$r_\infty = 100\ Mflop/s,$$
$$n_{1/2} = 100.$$

For a vector length of 100 elements we already achieve a computing rate of 50 Mflop/s, and for a vector length of 1000 we are almost at the asymptotic rate.

By the way, now it is nice to remember the story of Feynman's mechanical parallel computer in section 2.1.1. Remember how Bob Christy was impressed by the amount of problems that Feynman and his group could deliver every month, but how he had no idea of the real cycle time of the problems, as they went through the mechanical computer. In Hockney's languge, Bob Christy was impressed by the asymptotic compute rate r_∞, but had no idea of the half performance length $n_{½}$ and therefore had no idea of the actual performance of the pipeline.

4.4.4 The Parkbench Benchmark

Parkbench, an acronym from Parallel Kernels and Benchmarks, is a collection of benchmark programs, which will give an overall view of the performance of a parallel system. Refs. [45, 46] provide in-depth information on this benchmark suite. During the supercomputing '92 conference in Minneapolis a group was formed to establish a benchmark suite. This resulted in 1994 in Parkbench.

Parkbench is no longer actively used, but provides a very good example of how benchmarking of a parallel computer should be organized and was an inspiration for current benchmarking. For recent benchmarking efforts we refer to the Performance Database Server (at http://performance.netlib.org/performance/html/PDSreports.html) and to the Standard Performance Evaluation Corporation (SPEC, at http://www.specbench.org/) and then especially the High Performance Group (SPEC HPG at http://www.specbench.org/hpg/).

The suite consists of three types of benchmark programs, low level benchmarks, kernel benchmarks, and compact applications. The low-level benchmarks are used to measure basic performance of the processors and communication network. Parkbench also recommends to use LinPack and the Livermore loops to assess the speed of a single processor in the parallel computer. The kernel benchmarks are small applications that one usually encounters as kernel routines in full-blown production codes. By measuring the performance of the kernel benchmarks, one should already gain insight in the performance of production codes that are assembled from many of such kernel applications. Finally, the compact applications are small, but complete parallel applications. All benchmarks are written in the SPMD model, using Fortran77 with MPI.

The low level benchmarks contain single and multi-processor benchmarks, to measure timer resolution, performance of basic arithmetic operation, memory bottlenecks, and communication performance. The low level single processor benchmarks are:

Timer Resolution and Value: TICK1 and TICK2

To assess the resolution and accuracy of the measured wall clock times

Basis Arithmetic Operations: RINF1

A set of 17 common Fortran DO loops (such as Inner Product, Matrix Multiplication, DAXPY, etc.). Results are reported in terms of the (r_∞,$n_{½}$) metric.

Memory-Bottleneck Benchmarks: POLY1 and POLY2

Even if the vector lengths are long enough to overcome the vector startup overhead $n_{½}$, the peak rate of the arithmetic pipelines may not be realized because of the delays associated with obtaining data form cache or main memory. POLY1 and POLY2

quantify this dependence of computer performance on memory access bottlenecks. Define the computational intensity f of a DO loop as the number of floating point operations performed per memory reference to an element of a vector variable. It is observed that the asymptotic performance r_∞ increases as the computational intensity increases. This effect is characterized by two parameters;

$$(\hat{r}, f_{1/2}),$$

the peak hardware performance and the computational intensity to achieve half this rate. That is to say, the asymptotic performance is given by

$$r_\infty = \frac{\hat{r}_\infty}{1 + f_{1/2}/f}. \qquad [108]$$

POLY1 and POLY2 are polynomial evaluations by Horner's rule. POLY1 is a typical in-cache test, whereas POLY2 is an out-of-cache test of the memory bottleneck between off-chip memory and the arithmetic registers.

Table 11 gives examples of low level benchmark measurements on some microprocessors. All measurements were made with the highest level optimization for the compiler, and all data are in 64-bit precision. For more recent processors you should consult the benchmarking web pages [e.g. 44, 45].

Benchmark	**Intel i860XP 50 MHz**	**IBM RS 6000-530 25 MHz**	**DEC Alpha 133 MHz**	**Motorola PPC-601 80 MHz**
LinPack $n = 100$	14.7	9.54	20.7	12.8
Livermore Maximum	28.8	31.8	46.6	43.1
Livermore Minimum	2.62	1.34	4.47	4.89
RINF1 Only vector product				
r_∞	7.64		26.4	11.3
$n_{½}$	2.58		5.6	2.74
POLY1				
$\hat{r}_\infty$	13.5	25.9	88.9	
$f_{½}$	0.44	0.34	0.71	
POLY2				
$\hat{r}_\infty$	13.5	25.7		
$f_{½}$	1.12	0.91		

Table 11: Results for low level single processor benchmarks.

The multi-processor low-level benchmarks characterize the basic communication properties of a parallel computer. These low-level benchmarks are

Communication Benchmarks: COMMS1 and COMMS2

COMMS1 is a *ping-pong* experiment, COMMS2 is an *exchange* experiment. Throughput and setup times (latencies) are measured.

Total Saturation bandwidth: COMMS3

All-to-all communication experiment

Communication Bottleneck: POLY3

POLY3 assesses the severity of the communication bottleneck. It is the same as POLY1, except that the data for the polynomial evaluation is stored on a neighboring processor. The value of $f_{½}$ obtained therefore measures the ratio of arithmetic to communication performance. This can be compared with the fractional overheads we have discussed before.

Synchronization benchmarks: SYNCH1

Measures the time to execute a barrier synchronization as a function of the number of processors. The results can be quoted as barrier time or as barrier statements executed per second (barr/s).

The kernel benchmarks contain matrix benchmarks (dense matrix multiply, transpose operation, dense LU factorization, QR decomposition, matrix tridiagonalization), Fourier transforms (one – and three-dimensional), partial differential equation solvers (SOR kernel and a multigrid kernel), and some others (an embarrassingly parallel kernel, a conjugate gradient solver, a large intere sort, a parallel I/O benchmark).

The compact application benchmarks are typical of those found in research environments, and usually consist of up to a few thousand lines of source code. Compact applications are distinct from kernel application since they are capable of producing useful results. In many cases, compact applications are made up from several kernels, interspersed with data movements and I/O operations between the kernels.

A compact application benchmark must be a complete application, capable of producing results of research interest, be written in high quality code, i.e. extensively tested and validated and must be well documented and self checking. The application areas are climate modeling, computational Fluid Dynamics, finance, e.g. portfolio optimization, molecular Dynamics, plasma Physics, quantum Chemistry, quantum Chromodynamics, or reservoir modeling. Currently available compact application benchmarks are a parallel spectral transform shallow water model, and three computational fluid dynamics codes.

5 To Conclude

We started this introduction to parallel computing with HPC challenges and suggested that parallel computing is the way to go. A main conclusion is that *scalable parallel computing* is the answer to HPC challenges. We focussed on the SPMD paradigm of parallel computing, which is currently most suited for massively parallel computing. Within this paradigm domain decomposition is the essential parallelization strategy for the underlying complex systems. Parallel programming induces “new” issues, such as communication, load balancing, latency, etc.. In many cases a thorough redesign of “old” software is needed in order to obtain significance performance. Finally, when talking about performance, speedup alone seldom is a reliable metric. A good time complexity model is critical to the evaluation of the performance of a parallel program.

6 References

1 M.V. Wilkes and D.J. Wheeler, *The Preparation of Programs for an Electronic Digital Computer* (Addison-Wesley, Cambridge, MA, 1951).

2 See the official Top 500 Supercomputing sites list on http://www.top500.org/ .

3 M.V. Wilkes and W. Renwick, “The EDSAC, an electronic calculating machine,” J. Sci. Instrum. **26**, 385-391 (1949).

4 J.L. Hennesy and D.A Patterson, *Computer Architecture: A Quantitative Approach* (Morgan Kauffmann, 1990).

5 K. Hwang, *Advanced Computer Architecture, Parallelism, Scalability and Programmability* (McGraw-Hill, 1993).

6 D. Sima, T. Fountain and P. Kacsuk, *Advanced Computer Architecture, A Design Space Approach* (Addison Wesley, 1997).

7 R.W. Hockney and C.R. Jesshope, *Parallel Computers 2* (Adam Hilger, Bristol and Philadelphia, 1988).
8 D. Wallace, "Algorithms and Architectures for Grand Challenges in Physics," In *Very Large Scale Computation in the 21st Century* (Capital City Press, J.P. Mesirov Ed., Ch. 1, pp. 1-22, Montpelier, Vermont, 1991).
9 L.G. Richardson, *Weather Prediction by Numerical Process* (Cambridge University Press, 1922).
10 J. von Neumann, "A system of 29 states with a general transition rule," In *Theory of Self Reproducing Automata* (Illinois University Press, A.W. Burks Ed., Urbana, Illinois, paper was first published in 1952, 1966).
11 S.H. Unger, "A computer oriented towards spatial problems," Proc. Inst. Radio Eng. **46**, 1744-1750 (1958).
12 D.L. Slotnick, "Unconventional systems," in *AFIPS Conference Proceedings*, 1967, pp. 477 - 481.
13 D.L. Slotnick, "The fastest computer," Sci. Am. **224**, 76-87 (1971).
14 J.H. Holland, "A universal computer capable of executing an arbitrary number of subprograms simultaneously," in *Proc. East. Joint Comput. Conf.*, 1959, pp. 108 - 113.
15 M.C. Pease, "The indirect binary n-cube microprocessor array," IEEE Trans. Comput. **C-26**, 458 - 473 (1977).
16 G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors 1: General techniques and Regular Problems* (Prentice-Hall, Englewood Cliffs, New Jersey, 1988).
17 F. Darema, D.A. George, V.A. Norton, and G.F. Pfister, "A single-program-multiple-data computational model for EPEX/FORTRAN," Parallel Computing **7**, 11-24 (1988).
18 NSF, *Grand Challenge: High Performance Computing and Communications*, Report, Committee on Physical, Mathematical, and Engineering Sciences, Washington D.C.: US Office of Science and Technology Policy, National Science Foundation, 1992.
19 I. Danesh, "Physical Limitations of a Computer," Computer Architecture News **21**, 40-45 (1993).
20 U. Ghoshal and T. Van Duzer, "Superconductivity researchers seek to remove computational bottle-necks," Computers in Physics **6**, 585 - 592 (1992).
21 H. Kashiwagi, "Japanese super-speed computer project," In *High Speed Computing* (Springer Verlag, J.S. Kowalik Ed., pp. 117 - 125, Berlin, 1984).
22 S. Kotani, A. Inoue, T. Imamura, and S. Hassuo, Dig. Tech. Papers, 148 - 149 (1990).
23 G. Taube, "Is the Third Time a Charm for A Superconducting Computer?," Science **261**, 1670 - 1671 (1993).
24 K.K. Likharev and V.K. Semenov, IEEE Trans. Appl. Supercond. **1**, 3 - 28 (1991).
25 A.D. McAulay, "Researchers look to optics to move computer technology forward," Computers in Physics **6**, 594 - 602 (1992).
26 S. Lloyd, "A Potentially Realizable Quantum Computer," Science **261**, 1569 - 1571 (1993).
27 L.M. Adleman, "Computing with DNA," Scientific American, September, 34 – 41 (1998).
28 See e.g. http://www.beowulf.org/ or http://cnls-www.lanl.gov/Internal/Computing/Avalon/.
29 A.J. van der Steen and J.J. Dongarra, "Overview of Recent Supercomputers, http://www.phys.uu.nl/~steen/web01/overview01.html
30 G.C. Fox, R.D. Williams, and P.C. Messina, *Parallel Computing Works* (Morgan Kaufmann, San Fransisco, 1994). This book is also available on the web, http://www.npac.syr.edu/copywrite/pcw/ .
31 A.Y.H. Zomaya (Ed.), *Parallel & Distributed Computing Handbook* (McGraw-Hill, 1996).
32 M.J. Flynn, "Some computer organisations and their effectiveness," IEEE Trans. Comput. **C-21**, 948 - 960 (1972).
33 S. Raine, *Virtual Shared Memory: A Survey of Techniques and Systems,* Tech. Rept. CSTR-92-36, University of Bristol, Computer Science Department, 1992.
34 See the PVM web site, http://www.epm.ornl.gov/pvm/.
35 Nicholas Giordano, "The physics of vibrating strings", Computers in Physics **12**, 138-145 (1998).
36 W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C, The Art of Scientific Computing* (Cambridge University Press, 1988)
37 J. Stoer and R. Bullirsch, *Introduction to Numerical Analysis* (Springer Verlag, 1980).
38 D.H. Bailey, Supercomputer **45**, VIII-5, 1991. Technical Report: Nasa Ames: RNR-92-005, and http://science.nas.nasa.gov/Pubs/TechReports/RNRreports/dbailey/RNR-92-005/RNR-92-005.html.

39 G. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proc. AFIPS Conference*, 1967, pp. 483 - 485.
40 J.L. Gustafson, “Re-evaluating Amdahl's Law,” Communications of the ACM **31**, 532 - 533 (1988).
41 X.H. Sun and L.M. Ni, “Scalable Problems and Memory-Bounded Speedup,” J. Parallel Distrib. Comput. **19**, 27-37 (1993).
42 R. Hockney, *The Science of Computer Benchmarking*, ISBN 0-89871-363-3 (SIAM, 1996)
43 http://www.specbench.org/
44 http://performance.netlib.org/performance/html/PDStop.html
45 http://www.netlib.org/parkbench
46 Special Section in Scientific Programming, Volume 3, Number 2, Summer 1994.