Northeast Parallel Architectures Center

Issues in Parallel Computing

Gregor von Laszewski
gregor@nova.npac.syr.edu

Technical Report

SCCS 577

Science and Technology Center
111 College Place
Syracuse, NY 13244-4100
Tel.: (315) 443-1722, 1723; Fax: (315) 443-1973

Contents

Chapter 1

Supercomputer

1.1 Introduction

Existing computational problems challenge current research in the design of new computers and the development of new software technology to handle these machines. We show some important issues connected with these machines and their efficient programming.

To answer the question why one needs for example supercomputers we deal with two problems: *speed* and *size*. With the help of parallel computers one can for example increase the speed a job is executed or increase the problem while maintaining the same speed.

For example[17]:

- speed — sorting 10 million 64 bit integers takes 200 seconds on a DEC5000/200 serial workstation and only 0.2 seconds on a CM-5 with 1024 processors.
- Size — a 1024 node CM-5 has 32 gigabytes of memory (32 megabytes per note) so that larger problems can be solved.

First, we try to define the term Supercomputer and show problems in determining their performance. We give some simplistic performance measurements for Supercomputers and up-to-date workstations.

Second, we introduce the taxonomy for the performance analysis.

Than we use a simple but sufficiently complex problem, the graph bisection, in order to show how we can design programs

- sequentially
- with message passing
- and in data parallel fashion.

Finally, we show some sample applications and the problems inherented with their parallelization. Applications we are considering are embarrassingly parallel problems, regular problems, long and short range problems, irregular problems, and matrix problems.

We come to the conclusion that in certain cases it is easy to write a dataparallel program but in other cases it is from advantage to use message passing. The Algorithms and architectures designed for the future should therefore be able to allow both strategies in such a fashion that part of the machine might run a dataparallel program while others use message passing. This would enable the algorithm designer to concentrate on the problem and use the language design most suitable to solve this problem.

1.2 Supercomputer

There exists no precise definition of a Supercomputer. We define the class of Supercomputers as the fastest most powerful computers available. Due to the rapid progress in the VLSI design the performance of a supercomputers available in the sixties is today out-performed by personal computers. Today's Supercomputer have one or more of the following characteristics[56]:

- fast central registers
- multiple CPU's
- multi-staged (pipelined) functional units
- fast and large memories
- low communication latency between functional units

- vector and/or an array of processors
- huge amount of software
- high performance

Of these characteristics the high performance is the most important one. Often it is not clear how one can compare two different supercomputers with such different characteristics. In the next section we show elementary definitions used for benchmarking such machines. This is done in order to answer the question if we can compare different machines with each other.

Chapter 2

Computer Benchmarks

2.1 Benchmarks

Many vendors provide simple numbers to convince their customers to purchase a particular machine. To take all illusion immediately:

> There exist no perfect and simple measurement of the performance of a computer system.

Different companies and users evaluate the performance of a computer with different criteria. While some select the machine which solves their problem the fastest others include in their decision whether the system is cost efficient. Often it is important that previous purchased and developed software runs on these machines because of the cost efficiency. In this section we want to introduce some of the main terms used in benchmarking.
First, we show what parts are involved in the performance evaluation. Second, different performance metrics are introduced and problems inherented with them are shown. Measuring tools are introduced in order to provide practical help while gathering information about execution times and the number of times a statement is executed. Next, we introduce Runtime benchmarks and make the user aware who easy it is to “fool the masses” with benchmarks. Finally, we provide some data gathered from different sources.

2.1.1 Performance Evaluation

The Evaluation of the performance of a computer consists of two parts (Figure 2.1):

- Performance Analysis
- and Performance Measurement

The *Performance Analysis* is similar to mathematical analysis. Here a model theory is formulated to analyze the performance of the machine in order to determine how to use the system efficiently.
The *Performance Measurement* is an empirical process of gathering data in order to measure the performance of a real hardware system. Often it is difficult to draw conclusions from one benchmark generated for a particular program to another program. This means the benchmark is generally only valid for the particular program. For other programs or other input parameters the performance might look completely different.

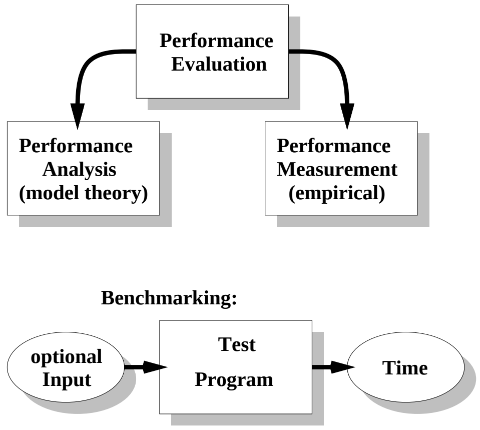

Performance Evaluation
Performance Analysis (model theory)
Performance Measurement (empirical)
Benchmarking:
optional Input
Test Program
Time

Figure 2.1: Performance Evaluation

2.2 Performance Metrics

In mathematics a metric is a well defined term. Metrics allow us to measure and order a quantity. In the theory of metrics a metric function d is defined on a metric space X mapping its arguments to X. Let $a, b \in X$, then

$$d(A, B) = d(B, A) \tag{2.1}$$

$$d(A + B) <= d(A) + d(B) \tag{2.2}$$

Unfortunately, we cannot apply the same rules to performance metrics, since mathematical inferences are not consistent. In addition, not all mathematical operators are applicable. A benchmark for a problem size of n elements might not be suited for problem sizes of $n-1$ or $n+1$. In practise this can be observed especially when dealing with powers of two due to hardware properties or with powers of ten due to software properties.
To develop a meaningful metric for benchmarking we should at least guarantee the following properties:

1. Reproducibility
2. Accuracy
3. Detailed

To achieve this goal most benchmarks are run in in stand-alone conditions not including problems occurring while using time sharing mode.
The *Time* is certainly the best measure because it fulfills all requirements mentioned above. All other measures are more or less inaccurate. The reader should note that it is also important to know in which context the time is taken. It is certainly unfair to claim that one machine is faster than the other, when one machine a highly optimized machine code is executed and on the other a not optimized Pascal code.
Often one finds the common view that a CPU determines the speed of a computation. This is very often not the case since for example IO slows down the computation a lot. A system can only be as fast as it is hindered by its slowest component (bottleneck).

2.2.1 Simple Metrics

Under less reliable measures we find [59]:

MIPS, GIPS, TIPS : Million (Giga, billions; tera, trillion) Instructions Per Second. This Measure is used for marketing but does not indicate the performance of the machine.

Instruction : An instruction is an event, frequently a change in the state of a CPU. Often, an instruction is synonymous with the clock rate of a machine ignoring instructions requiring more than one clock pulse tick to execute.

MFLOPS, GFLOPS, TFLOPS : Million (Giga, billions; tera, trillion) Floating-Point Operations Per Second it ignores non-floating-point instructions. This measure is e.g. particular bad if one abstracts from 2 dimensional arrays to 3 dimensional arrays due to additional amount in the address calculation.

Packets Per Second : Unit of measure used by networking and communications community.

MHz, GHz, Bits per Second, Bytes per Second, Words per Second : These measures are often used to missmeasure the performance of computer networks like Ethernet (tm). It confuses the base band carrier frequency with the data transfer rate.

2.2.2 Normalized Metrics

Under a normalized we understand a metric which uses some kind of test which result is included in a weighted fashion into an overall metric measure. Many tests on one problem might be done in order to obtain a particular point in the metric space. Examples for this kind of metric is the SPEC benchmarking site which we will describe later in more detail.

2.2.3 Temporal Performance

If one is interested to compare the performance of different algorithms for the solution of the same problem the *temporal performance* is a useful measure.

It is defined as the inverse to the execution time. The units of the temporal performance are given in solutions per second or timesteps per second.
The *simulation performance* is a special case of the temporal performance measuring the simulation of a certain time period of physical time. This performance measure is useful in physical problems, e.g. climate modeling or weather forecasting.

2.3 Measurement Tools and Environments

To use existing tools for benchmarking and performance evaluation one has to understand the following terms [26]:

- User time
- System time
- CPU time
- Elapsed time

A computer runs generally in two modes, the user and the kernel mode. During the user mode the instructions of the program are executed. Instructions which invoke the kernel are executed in kernel mode. An example of such instructions are I/O operations. The time spend in the two modes is kept separately in user time and system time. The CPU time is the sum of system and user time. Under elapsed time we understand the actual wallclock time that passed since the calculation is in progress. Timesharing, I/O operations, paging, swapping, and bottleneck in the memory bandwidth are reasons for larger elapsed times in contrast to the CPU time.
On many machines program profiling tools are available. Most common are the programs prof, gprof and tcov. Prof produces an execution profile of a program. For each external symbol, the percentage of time spent executing between that symbol and the next is printed, together with the number of times a routine was called and the number of milliseconds per call. Due to the overhead of the housekeeping functions the Numbers might not represent the actual run. Tcov produces a test coverage analysis and statement-by-statement profile of a

program. This means that for each statement a counter is provided to determine the number of executions of this statement while running the program.
On a Cray we find for example flowtrace, hpm, on an SGI/MIPS gr_osview, hinv, pixie, and on a Convex syspic. For parallel machines like the CM-5 we have PRISM and for the nCube and iPSC machines there exists also debuggers and graphical profiles. Usefull tools are mentioned for example in in [20, 33, 34, 58, 57, 43, 69, 19, 15, 4]

2.4 Common Runtime Benchmarks

2.4.1 Linpack LU Decomposition

The LINPACK benchmark consists of a simple LU decomposition of a dense linear system (Gaussian elimination) using the LINPACK library [23, 10, 9]. To exclude the favor of some architectures for problems whose size is a multiple of the power of 2 the problem sizes 100x100,300x300, and 1000x1000 are used.
The advantages of this benchmark is that it is

- a short benchmark written in simple fairly portable FORTRAN.
- updated quite often.
- without charge electronically available.

The disadvantages are also obvious. Because of the problem structure it is not easily parallelizable, the parallelization is dependent on the machine so that the portability is lost [78, 72]. Some numbers in the LINPACK report are produced by unpublished assembly language or Fortran codings tuned for very specific computer models.

2.4.2 SPEC Benchmark

SPEC, the Standard Performance Evaluation Corporation, is formed to "establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers" (SPEC's bylaws). The organization develops suites of benchmarks intended to measure computer performance. The results are published in a report available quarterly.

Since we already learned that no benchmark can fully characterize the overall system performance, the group combines results of a variety of realistic benchmarks to find the expected real performance.

CPU Benchmarks

"There are currently two suites for floating point and integer calculations, measuring the performance of CPU, memory system, and compiler code generation. They normally use UNIX as the implementation platform, but they have been ported to other operating systems as well. The percentage of time spent in operating system and I/O functions is generally negligible".[5]

CINT92 The integer benchmark suite contains 6 benchmarks. All of them are written in C from application areas such as Logic Design, Compiler, Interpreter, Data Compression and Spreadsheet programs.

CFP92 The floating-point benchmark suite contains 14 programs, from which 12 are written in Fortran, 2 in C. The individual programs are from the areas of Circuit Design, Simulation, Quantum Chemistry, Electromagnetism, Geometric Translation, Optics, Robotics, Medical Simulation, Quantum Chemistry, Simulation, Quantum Physics, Astrophysics, and NASA Kernels.

The CPU benchmarks can be used to measure the speed and the throughput.

Speed Measurement

The speed measurements are normed in respect to the execution time on a VAX 11/780. This "SPEC Ratio" is available for each individual benchmark. They are defined as the ratio of the wall clock time to execute one single copy of the benchmark, compared to the execution time on a VAX 11/780.
Looking into the publications one will easily recognize that the different SPEC ratios for a given machine can vary widely. Therefore, it is important to compare the characteristics of the own application area with the individual SPEC benchmarks to consider those benchmarks that best approximate the job. Sometimes it might be useful just to argue with an average value, which is expressed in the following numbers:

SPECint92 = geometric average of the 6 SPEC ratios from CINT92

SPECfp92 = geometric average of the 14 SPEC ratios from CFP92

SPECint92 can be used to estimate a machine's single-tasking performance on integer code while SPECfp 92 can be used to estimate a machine's single-tasking performance on floating-point code.

Throughput Measurement

With this measurement method, called the "homogeneous capacity method", several copies of a given benchmark are executed. This method is particularly suitable for multiprocessor systems. The results, called SPEC rate tell how many jobs of a particular type can be run in a given time which is in this case a week. As before the execution times are normalized with respect to a VAX 11/780. Therefore, the SPEC rates characterize the capacity of a system for compute-intensive jobs of similar characteristics.
Similar to the speed metric, the following average values are defined:

SPECrate_int92 = geometric average of the 6 SPEC rates from CINT92

SPECrate_fp92 = geometric average of the 14 SPEC rates from CFP92

The integer SPECrate can be used to estimate a machine's overall multi-tasking throughput for integer code, while the Floating-Point SPECrate can be used to estimate a machine's overall multi-tasking throughput for floating-point code. As an example, the Table 2.1 and the Figure 2.2 show some floating point and integer SPEC marks published recently for different workstations.

2.4.3 Parallel Benchmarks

So far we introduced classical sequential benchmarks. These benchmarks are important for evaluating the performance of a single processing node which might be used in an MIMD machine. In order to evaluate other hardware properties like the communication network not only the arithmetic speed is measured but also the communication speed.

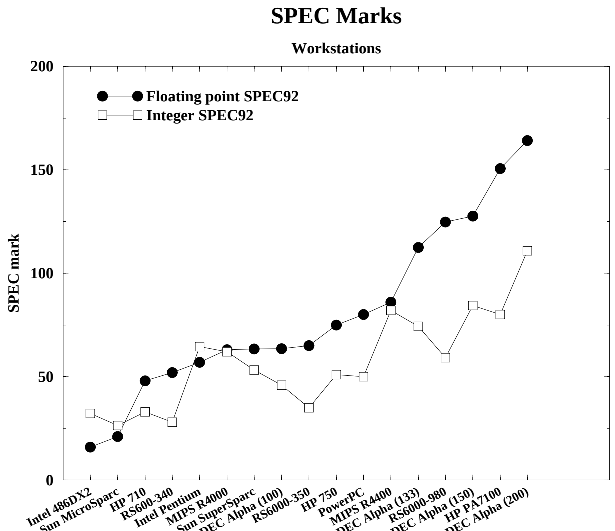


Figure 2.2: SPEC marks for different sequential computers and workstations

One of the most popular communication benchmarks is the so called *ping-pong* measure, the basic communication speed in an MIMD machine while sending a message from one processor to the other and back.
Other benchmarks might be useful to determine the total saturation bandwidth and the communication bottleneck as described in [46].
Beside this benchmarks for parallel computers the arithmetic benchmarks as given by Linpackd, NAS benchmark, and other benchmarks from Livermore National are often used.
The aim for a parallel benchmark suite is to provide benchmarks to evaluate performance on a wide range of levels from simple machine parameters to complex applications. In the applications memory , communications, IO and other bottlenecks might become important.

Table 2.1: SPEC mark ratings of Workstations

From Sterlin Report on Parallel Processing, Issue71, April 93 [70]

Machine	fp SPEC mark	int SPEC mark	MHz
Intel 468DX2	16	32.2	33
SUN MicroSparc	21	26.4	50
HP 710	48	33	50
RS6000-340	52	28	33
Intel Pentium	56.9	64.5	66
MIPS R4000	63	62	50
SUN SuperSparc	63.4	53.2	40
DEC Alpha	63.6	45.9	100
RS600-350	65	35	41
HP750	75	51	66
PowerPC	80	50	66
MIPS R4400	86	82	75
DEC ALpha	112.5	74.4	133
RS6000-980	124.8	59.2	62.5
DEA Alpha	127.7	84.4	150
HP PA7100	150.6	80.0	99
DEC Alpha	164.1	110.9	200

2.5 Ways to Fool the Masses with Benchmarks

To make us aware that benchmarks are often misused we give the 12 ways to fool the masses without further comment, published by David Bailey et al. [13]:

1. “Quote only 32-bit performance results, not 64-bit results”.
2. “Present performance figures for an inner kernel as the performance of the entire application”.
3. “Quietly employ assembly code and other low-level language constructs”.
4. “Scale up the problem size with the number of processors, but omit any mention of this fact”.
5. “Quote performance results projected to a full system”.
6. “Compare your results against scalar, unoptimized code on Crays”.

7. “When direct run time comparisons are required, compare with an old code on an obsolete system”.
8. “If megaFLOPS rates must be quoted, base the operation count on the parallel implementation, not on the best sequential implementation”.
9. “Quote performance in terms of processor utilization, parallel speedup or megaFLOPS per dollar”.
10. “Multilate the algorithm used in the parallel implementation to match the architecture”.
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 videos, and don’t talk about performance”.

2.6 Supercomputer Performance

In this section we summarize some data available for parallel computers.

The Performance is given for 64 bit floating point arithmetic, unless otherwise stated. Values noted with (1)are either 32 or 36 bit. For the earlier machines, the flop rate is taken from the average of an add and a multiply.

For parallel machines, values are listed for both the maximum size machine which has been built, and for the size of a scaled up machine (scaled in powers of 2) which is closest in price to $20M (i.e. supercomputer price), even though it may not exist (this is denoted by a (2)). For this virtual machine, the performance figure given in brackets is the value scaled to an exactly $20M machine. This is the point which appears in the plot.

For the vector machines, values in brackets in the “procs x pipes” column are number of Flops per processor per clock cycle, if that is not equal to the number of pipes. The performance is given by: Max Flops = Number of processors x Flops per cycle x Cycles per second.

Best performance is a value that has been achieved by a real applications code or benchmark (LP = LINPACK, MM = matrix multiplication).

The performance of sequential and supercomputers are displayed in Figure 2.3.

Table 2.2: Sequential Scalar Mainframes

Machine	date	price $M	processor	max perf kflops		refs
ACCOUNTING MACHINES (Harvard-IBM)	44		electro-mechanical	0.002		[63, 68]
ENIAC	46Q1	0.5	vac tubes	0.6 (1) fixed pt	5 (add) (1) 0.35 (mult) (1)	[63, 68, 16]
SEAC	50Q2?		vac tubes	??	??	[68, 16]
UNIVAC 1	51Q2	0.4?	vac tubes	0.7 (1) fixed pt	2 (add) (1) 0.45 (mult) (1)	[63, 68, 16]
MANIAC	52		vac tubes	??	?? (1)	[16]
IBM 701	53Q2		vac tubes 32 kHz	3.5 (1) fixed pt	16 (add) (1) 2 (mult) (1)	[63, 68]
IBM 704	55		vac tubes 84 kHz	10? (1) fixed pt 7 (1) floating pt	42 (add) (1) 6?(mult) (1) 12 (add) (1) 5 (mult) (1)	[63, 68] [16]
IBM 7090	59Q4	3	transistors 460 kHz	58 (1) floating pt	230 (add) 33 (mult)	[63, 68]

Table 2.3: Multi-Unit Scalar or Pipelined Scalar Machines

Machine	date	price $M	clock MHz	scalar Mflops		max perf Mflops	refs
IBM STRETCH (IBM 7030)	61	8	1?	0.7	1.0 (add) 0.55 (mult)		[63, 68, 16]
CDC 6600	64Q4		10	1.5	3.3 (add) 1.0 (mult) (3)	3.0 ?	[47, 16]
CDC 7600	69		36	5.0??		10??	[47]

(3)(2 mults and 3 adds every 10 cycles. In practice, about 2.5 flops/cycle)

2.6.1 Some Machines

The CM5

The CM5 can operate in both SIMD and MIMD mode. The available machine has 32 nodes each with 32 megabyte of memory. Each node includes a RISC processor as well as four vector units capable of 128 MFLOP peak performance. The RISC processors are 33MHz SPARC processors. They use 64 KByte cache for instructions and data together. The Processor is rated with 22 Mips and 5 MFLOPS.

The interconnection network between processing elements is given by a fattree shown in Figure 2.4. The CMMD message passing library provides the above mentioned sending and receiving commands.

Table 2.4: Modestly Parallel Pipelined Vector Machines

Machine	date	price $M	procs x pipes	clock MHz	max perf Gflops	best perf Gflops	refs
CRAY 1S	76	8?	1 x 2	80	0.16	0.11 LP	[1, 2, 47]
CDC CYBER 205	81		1 x 4 (8?)	50	0.40	0.20 LP	[1, 2, 47]
CRAY XMP/2 XMP/4	83Q2 85 86Q3	 25? 22	2 x 2 4 x 2 4 x 2	105 105 118	0.42 0.84 0.94	 0.82 LP	[1, 2, 71] [1, 2, 3]
CRAY 2S	85Q2	20	4 x 2	244	2.0	1.7 MM	[1, 2]
FUJITSU VP200	83Q4		1 x 3 (4) ?	133	0.53		[2]
HITACHI S-810	83Q4		1 x 6?	71	0.43?		[2]
HITACHI S-810 ?	84 ?		1 x 6	143	0.84		[1, 2]
FUJITSU VP400 ?	85		1 x 3 (8)	143	1.14		[1, 2]
NEC SX-2	86		1 x 16 (8)	167	1.3		[1, 2]
IBM 3090VF	86	10	6 x 2	69	0.82	0.54 LP	[1, 2]
HITACHI S-820	87Q4		1 x 8 (12)	250	3.0		[1, 2, 3]
ETA 10E	88Q1?	22	8 x 2 (9)	95	6.9		[1, 2, 3]
CRAY YMP	88Q4	23	8 x 2	167	2.7	2.1 LP	[1, 2, 3]
NEC SX-3	90Q4	24	4 x 16	345	22	20.0 LP	[1, 3]
FUJITSU VP2600	91Q1	15	2? x 2 (8)?	312	5.0	4.0 LP	[1, 2]
CRAY C-90	92Q1?	30	16 x 4	250	16	13.7 LP	[1, 3]
HITACHI S-3800	93?		4 x ? (16)	500	32		
FUJITSU VPP500	93Q4?	19 125	19 x 2 222 x 2	800	30 350	1.6 per processor	

The iPSC\860

The iPSC\860 is a MIMD machine. The available machine has 16 nodes each with 8 Mbyte of memory. Each node includes a 80860 processor and a direct-connect module which controls eight bidirectional communication channels. The channels are rated with a peak performance of 2.8 Mbyte per second. The maximum size of a Hypercube can hold up to 128 nodes.

The interconnection network between processing elements is given by a Hypercube shown in Figure 2.4. Libraries for the message passing provide routines similarly to the above mentioned sending and receiving commands.

Intel Touchstone

The Intel Touchstone Delta system is a message-passing multicomputer, consisting of an ensemble of individual and autonomous nodes that communicate across a two-dimensional mesh interconnection network (Figure 2.4). It has 513 computational i860 nodes, each with 16 Mbytes of memory and each node has a peak speed of 60 double-precision Mflops, 80 single-precision Mflops at 40 MHz. A Concurrent File System (CFS) is attached to the nodes with a total of 95 Gbytes of formatted disk space. The operating system is Intel's Node Executive for the mesh (NX/M).

Table 2.5: Massively Parallel Machines

Machine	date	price $M	nodes	clock MHz	max perf Gflops	best perf Gflops	refs
ICL DAP	80	2 (1M+host) 17 (2)	4K bit 64K bit	4	0.02 (1) 0.005 0.08 (0.1)	.017 MM	[71, 47]
Goodyear MPP ?	83Q1	4 16	16K bit (2)64K bit	10	0.3 (1) 0.08 0.3 (0.38)		[47]
Intel iPSC/1	85Q3	1 16	128 80286/7 (2)2K 80286/7	?	0.008 0.13 (0.16)		[2, 71, 3]
nCUBE-1	86	1.7 14	1K (2)8K	8	0.3 0.5 (1) 2.4 (3.4)		[2, 71]
TMC CM-2	87Q3	10	2K Weitek	7	14 28 (1)	10.4 LP	[1, 71]
Intel iPSC/2 VX	87Q4	4 32	128 Weitek (2)1K Weitek	10?	0.85 2.56 (1) 6.8 (4.2)		[2, 71]
Transputer	88	4? 30?	400 T800 (2)8K T800	25	0.4 0.6 (1) 8.0 (5.3)		[1, 2, 71]
nCUBE-2	90	3.8 30	1K (2)8K	20	2.4 3.4 (1) 19 (12.5)	1.9 LP	[1, 71]
Maspar MP-1	90Q1	1 30	16K (2)512K	2.5	0.58 1.3 (1) 19 (12.5)	.44 LP	[1, 71, 3]
TMC CM-5	92Q4	25	1K x 4 VUs	32	130		[3]
Intel Paragon	92Q4	30?	2K 860XP	50	150 300 (1)		[3]
Maspar MP-2	92Q4	1.7 27	16K (2)256K	2.5	2.4 6.4 (1) 38 (29)		[3]
KSR-1	92	30	32 (2)1088	20	1.3 43 (29)		[3]
Meiko			400				
X nCUBE-3							

Scalable Architecture

We call computer architecture scalable if it can be either used for the design of an arbitrarily large machine or which increases its performance linear in the amount of hardware investment.

Under FLOPS we do understand the number of floating point operations per second.

Distributed (heterogeneous) Computer Architecture

In contrast to Supercomputers which are assumed to be one entity a distributed computer consist of many independent computers. The are connected over a network. A network of workstations is an example for distributed computer. Since nearly every site has such resources available, it is a cost-effective alternative approach to solve a problem with existing hardware. If the entities are of different computational power we have a heterogeneous distributed computer. In the tables shown previously we did not include performance data for distributed systems.

		Machine	available	Number
1.9		Cray Y-MP4/2-64, 128		
1.9		Cray Y-MP8/2-64		
1.91		Intel iPSC/860 128		
2	?	NEC SX-3		
2.01		TMC CM-200/64K		
2.24	?	Intel Paragon XP/S-150	3Q93	
2.26		MasPar-2208		
2.28		TMC CM-5/32VU		
2.31		Hitachi S-820/60		
2.33		Cray M94-256		
2.33		Cray M98/4-1024		
2.41	?	Cray T3D/AC64	3Q94	
2.41	?	IBM GF-11		
2.48		KSR1-48		
2.51	?	Babaian Elbrus-3		
2.51	?	SRI MCS Supermacroneurocomputer		
2.64	?	Intel Paragon XP/S-10	3Q93	
2.72		Fujitsu VP2400/20		
2.78		Cray Y-MP4/3-32, 64		
2.78		Cray Y-MP8/3-64		
2.8	?	Fujitsu AP1000/256		
2.9		KSR1-56		
3.08		Hitachi S-820/80		
3.14	?	IBM SPP-64		
3.31		KSR1-64		
3.4		Fujitsu VPX240/20		
3.4		Fujitsu VPX260/10		
3.4		Siemens-Nixdorf S400/10		
3.4		Siemens-Nixdorf VP-400/EX		
3.53		nCUBE2/1024		
3.7	?	Cray 2S/8-128		
3.7		Cray Y-MP/4-32, 64		
3.7		Cray Y-MP8/4-132, 64, 128		
3.77	?	Intel Paragon XP/L-15		
3.8	?	Cray 4/1	4Q93	
4.02	?	Toshiba TS/1-64	2Q95	
4.3		Cray C92A-128	4Q93	
4.3		Cray C94/2-64	3Q93	
4.5		TMC CM-5/64VU-32/256		
4.51		DECmpp/SX 200/16K		
4.51		MasPar-2216		
4.64	?	NEC SX-3/21R		
4.65		Cray M98-4096		
4.69	?	IBM RS/6000	3Q93	256 *
4.76	?	NEC Cenju-3/64		
4.83	?	Cray T3D/128	1Q94	
5.03	?	Adaptive Solutions CNAPS-256		
5.21		Cray Y-MP8/6-32, 64, 128		
5.44		Fujitsu VP2600/10, 20		
5.6	?	Fujitsu AP1000/512		
5.89		KSR1-128		
5.97	?	Intel Paragon XP/L-29		
6.95		Cray Y-MP/8-32, 64, 128, 256		
7.6	?	Cray 3/4-128		

		Machine	available	Number
8	?	NEC SX-3/14		
8	?	NEC SX-3/22		
8.04	?	Siemens-Nixdorf VPP500/20	4Q94	
8.25		Cray C94A-128	3Q93	
8.7	?	Hitachi S-3800/480		
8.8	?	Intel Paragon XP/L-35		
9.1		TMC CM-5/128VU-32/256		
9.28	?	NEC SX-3/41R		
9.39	?	IBM RS/6000	4Q93	512 *
9.65	?	Cray T3D/256	3Q93	
10.05	?	Multicomputer 16K/40		
10.88	?	Fujitsu-NEC VP2600/40		
11.2	?	Fujitsu AP1000/1024	4Q93	
11.78		KSR1-256		
13.6		Siemens-Nixdorf S400/40		
13.6		Siemens-Nixdorf S600/20		
15.2	?	Cray 4/4	4Q94	
15.47	?	Intel Touchstone Delta 570		
15.6		Cray C916/8-128		
15.6		Cray C916/8-256		
15.6		Cray C98-128, 256, 512		
16	?	NEC SX-3/24		
17.76	?	RWC-1/1K	2Q96	
18.2		TMC CM-5/256		
18.56	?	NEC SX-3/24R		
18.85	?	Intel Paragon XP/L-75	4Q93	
19.04	?	NEC Cenju-3/256	2Q94	
19.32	?	Cray T3D/512	3Q93	
25.13	?	Alenia Spazio Ape 100 Quadrics Parallel		
25.13	?	IBM SPP-512	4Q94	
27.84	?	NEC SX-3/34R	4Q93	
27.91		Cray C916-256, 512		
32	?	NEC SX-3/44		
35.04		TMC CM-5/512		
35.18	?	Intel Paragon XP/L-140		
37.12	?	NEC SX-3/44R		
37.22		TMC CM-5/544VU		
37.7	?	Intel Paragon XP/L-150	2Q94	
38.6	?	Cray T3D/1024	4Q94	
50.26	?	Meiko CS-2/256-512VU	1Q94	
59.31	?	NAL NWT/140	??	
64.34	?	Toshiba TS/1-1024	2Q96	
70.07		TMC CM-5/1024	??	
75.39	?	Hitachi CP-PACS/2048	1Q96	
98.26	?	Archipel Volvox-LHPC/2048	4Q94	
250.91	?	Cray T3D/6656	4Q96	
284.16	?	RWC-1/16K	4Q97	
494.21	?	Intel?-2048	4Q97	
1136.64	?	RWC-1/64K	4Q98	
2513.16	?	Fujitsu?	4Q99	

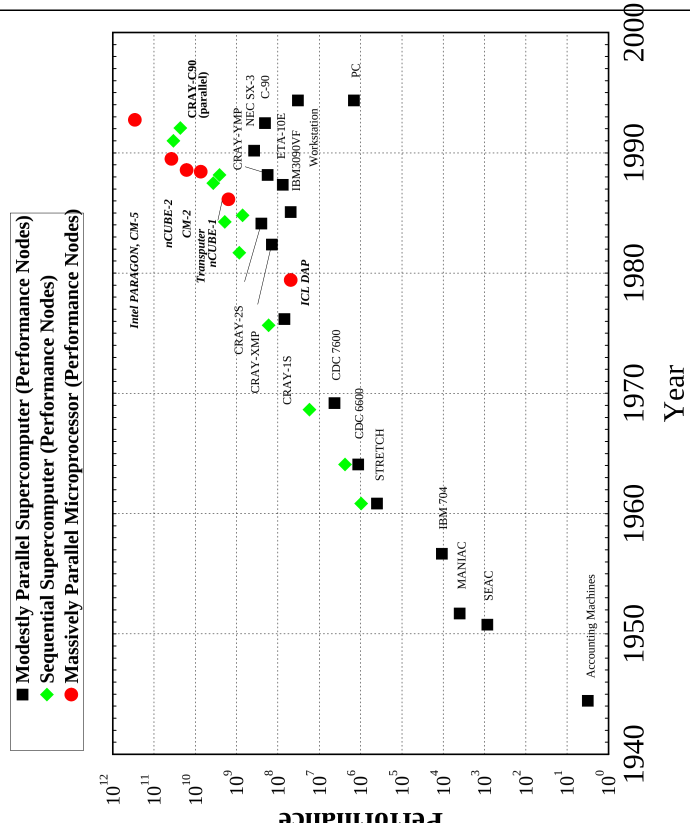


Figure 2.3: Performance of different sequential and parallel Supercomputers

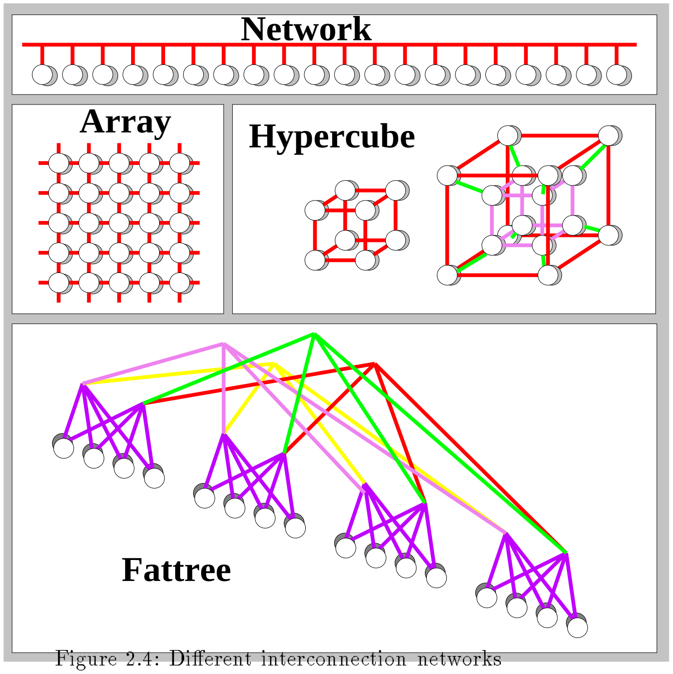


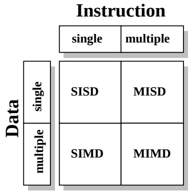
Figure 2.4: Different interconnection networks

Chapter 3

Taxonomy of Computers

3.1 Flynn's Classification

One common way to classify parallel computers is to use the taxonomy proposed by Flynn [30]. It is based on the observation that each computer whether parallel or sequential executes a stream of instructions on a stream of data. An instruction stream is a sequence of instructions as executed by the machine. A data stream is a sequence of data including input and output, partial or temporary results. The data and instruction stream can be single or multiple so that four classes are distinguished as shown in Figure 3.1.



		Instruction	
		single	multiple
Data	single	SISD	MISD
	multiple	SIMD	MIMD

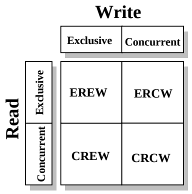
Figure 3.1: Classification by Flynn

Figures 3.3-3.6 show the relation of data and instruction streams to the processor of Flynn's taxonomy.

Examples for SIMD machines are architectures in which homogeneous processes execute the same instruction synchronously on their own data such as the CM-2 or the Maspar and architectures in which each operation may be executed on vectors of fixed or varying length. Examples for MIMD machines are the nCube and the iPSC delta. Sequential computers are SISD.

3.2 Parallel Random Access Machine

One of the most famous theoretical models for parallel computation is the Parallel Random Access Machine (PRAM). The model has an arbitrary, but finite, number of processors and an arbitrarily large shared memory which can accessed by the processors in random access mode. The processors are synchronized but can execute different instructions at a time. Shared memory computers can be divided into subclasses according to whether access to the memory is done concurrent or exclusive [7] as shown in Figure 3.2.



Read \ Write	Exclusive	Concurrent
Exclusive	EREW	ERCW
Concurrent	CREW	CRCW

Figure 3.2: Shared Memory SIMD Computers

EREW read and write on a memory location can be done only exclusively. (that means only one processor has access to the particular memory location)

CREW read can be performed concurrently

CRCW reads and writes can be performed concurrently. since processors can write in the same memory location at the same time a strategy has to be defined to do the writing. A simple one would be a random selection of the processor writing. Other writes at the same time step will be ignored.

3.3 SPMD

In many cases only a single program runs on the different processors of a parallel computer. This is abbreviated with SPMD or Single Program Multiple Data. We view SPMD either an extension of SIMD or a restriction to MIMD.
Other classification scheemes can be found in [28, 29, 37, 40, 41, 42, 45].

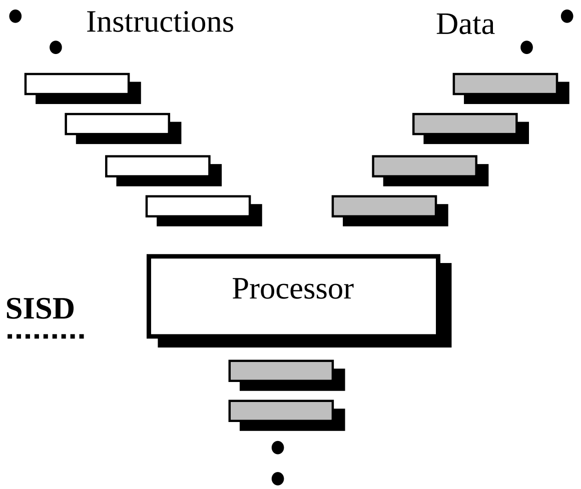


Figure 3.3: SISD

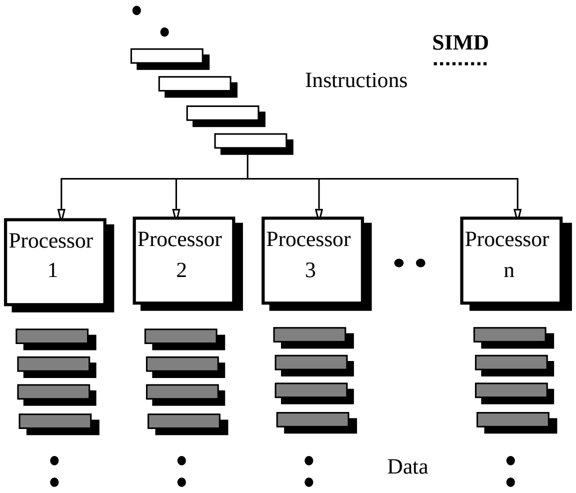


Figure 3.4: SIMD

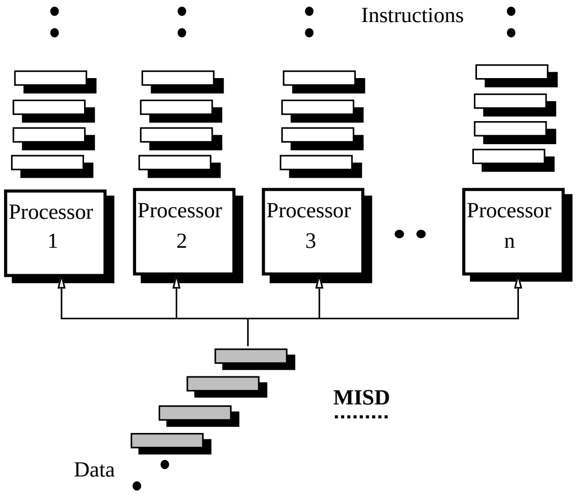


Figure 3.5: MISD

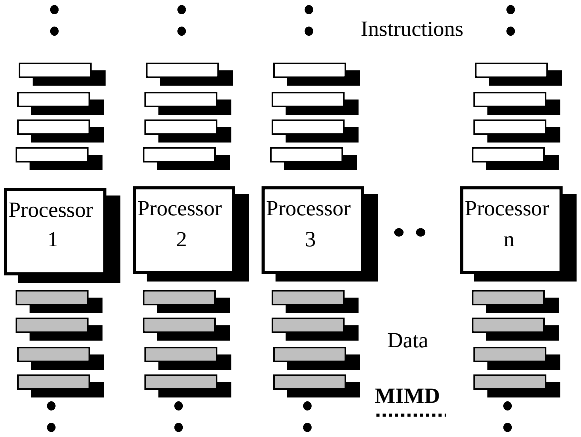


Figure 3.6: MIMD

Chapter 4

Memory Access

To write efficient algorithms (not only for multiprocessor computers but also for sequential machines), it is necessary to review the concept of a *memory hierarchy*.

Normally, the computation done in a central processing unit (CPU) is much faster than the time necessary to move the required data from the memory to the registers of the CPU. The process of moving the data is called *fetching* and the time required for transferring data from a part of the memory to the CPU is called *memory access time*. In order to use the processor efficiently it is important to keep the memory access time as small as possible. Unfortunately, it is too expensive to build very fast memories with sufficient capacity for scientific applications requiring huge amounts of data. Therefore, a *memory hierarchy* is used to decrease the cost of the memory system while retaining efficient memory access times. Figure 4.1 shows a typical memory hierarchy. The closer the memory level is to the registers of the processor the faster is the access.

For example, to use data stored in the external memory it has to pass through all levels of the memory hierarchy. Often, access time can be decreased if the usage of specific data can be predicted, so that data is transferred into a faster part of the hierarchy before it is actually referenced.

One simple way to evaluate if a program can make use of the hierarchy in an efficient way is to keep the ratio of operations to data movement as large as possible. This ratio is important to achieve high performance when exploiting concurrency.

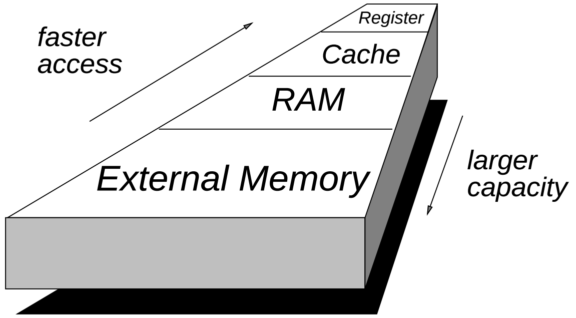


Figure 4.1: Typical memory hierarchy in a computer

For example, the following statement inside a loop performing matrix multiplication,

$$c_{ij} \leftarrow c_{ij} + a_{ik} * b_{kj}$$

requires three memory accesses to obtain the data c_{ij}, a_{ik}, b_{kj}, and one to store the result in c_{ij}. Addition and multiplication count as one floating point operation each. The ratio of floating point operations to memory access time is $r = \frac{1}{2}$.

A simple programming trick to improve this ratio is to figure out how data is stored in the memory. One has to know that most memory organizations use specific strategies to reduce the memory access time. A common rule on many machines is to fetch a block of data instead of only one datum at a time. The distance between elements in the memory is called *stride*.

Therefore, it is best to formulate the algorithms in such a way that data elements used in consecutive computation steps are stored in contiguous addresses of the memory. Hence they are fetched in a block requiring fewer memory accesses. Figure 4.2 shows how data (a matrix) is stored in a memory using the Fortran programming language. Having this in mind it is obvious why Fortran is called a column oriented programming language.

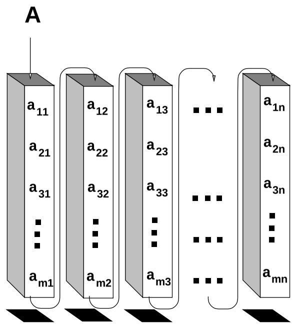


Figure 4.2: Storage of a two dimensional array in column oriented programming languages like Fortran

Under the assumption, that a machine is able to fetch α contiguous data elements from the memory in one time step, some statements of the loop performing the matrix multiplication steps can be rewritten as

$$c_{ij} \quad \leftarrow \quad c_{ij} + a_{i,k} * b_{k,j} + a_{i,k+1} * b_{k+1,j} + \cdots + a_{i,k+\alpha-1} * b_{k+\alpha-1,j} \quad (4.1)$$

This leads to 2α floating point operations, 2 memory accesses for storing and fetching c_{ij}, α memory accesses for fetching the a_{ik}'s and one memory access for all b_{kj}'s. The ratio is $r = \frac{2\alpha}{3+\alpha}$.
By storing the matrix **A** as its transpose, A^t, one can rewrite the multiplication as

$$c_{ij} \quad \leftarrow \quad c_{ij} + a^t_{k,i} * b_{k,j} + a^t_{k+1,i} * b_{k+1,j} + \cdots + a^t_{k+\alpha-1,i} * b_{k+\alpha-1,j} \quad (4.2)$$

where a^t_{ki} specifies the element in the k-th row and i-th column of A^t.
Now there is only one memory access necessary to fetch vector a^t $(a_{i,1}, \ldots, a_{i,k+\alpha-1})$.
Therefore, the ratio is $r = \frac{\alpha}{2}$. The prediction of a maximal vector length α depends on many factors: the machine used, the memory hierarchy, and their fetching algorithm. Algorithms which update a block of contiguous vectors instead of only one vector at a time are known as *blocked algorithm*. This way the

work is done *locally* on a block of data.
Numerical experiments [78, 60] showed that traditional linear algebra algorithms do not achieve high performance on distributed-memory multiprocessors because of the lack of data locality. Therefore, data locality is the fundamental problem in parallel computing and has great influence on the performance on such machines. The use of block based algorithms is one of the most efficient ways to improve the performance of numerical algorithms on distributed memory machines.

Chapter 5

Programming Models

In this chapter we concentrate on the use of different programming models. We consider

- sequential programming,
- data parallel programming,
- and message passing programming.

We develop the code for a real application for each of these models in order to outline their specific features. We show different algorithmic details while parallelizing the original sequential algorithm. The application problem we consider is known as *recursive bisection of points* in a plain.

Bisection of Points Recursive bisection is a very fast partitioning strategy. It is often used for dividing points in a two dimensional plane into a number of partitions containing an approximately equal number of points. The membership of a point to a specific partition is specified by its location in the two dimensional plane.
A realistic application of the algorithm is described in [31]. In some VLSI circuit simulations the computational core is dominated by the simulation of the transistors. To increase the speed of the simulation the goal is to distribute the calculations onto different processors of a parallel machine. To achieve load balancing the recursive bisection technique is used. On each processor one should map an equal number of transistors for the simulation to achieve a minimal execution time for the simulation.

5.1 Sequential Programming

5.1.1 Problem Analysis

The following example is used to illustrate the recursive bisection algorithm [76]. Assume that a number of random points are located in a two dimensional finite plane. The problem is to map an equal number of points on a given number of processors. Figure 5.1 shows a two dimensional plane with 100 random generated points.
First, the algorithm places half of the points to the left and the other half to the right. This can be done by sorting the elements in the x-dimension. In the next step the same is done in each of the parts separately in the y-dimension. This is repeated as long as the desired number of partitions is found. In the Figure 5.1 results for 2, 4, 8, and 16 partitions are given.
In order to keep the problem as simple as possible we assume that we are not interested in the cost generated while mapping the final result on different processor topologies. This is beyond the scope of this chapter.

5.1.2 Sequential Sorting

The problem analysis suggests that a fast sorting algorithm is the basis of the recursive bisection strategy. Quicksort is one of the most popular sequential sorting algorithms due to its good average time complexity of $O(n \log n)$ for large problem sizes. The quicksort algorithm is based on the important fact that exchanges are executed over large distances which leads to it's high efficiency.
First, the median of a sequence of data items is found. The median is the data item in the middle of the list. This partitions the initial sequence into two subsequences. The subsequences have the property that they are either larger or smaller than the median. This step is repeated recursively on the so generated subsequences till no further divide-and-conquer is possible. A pseudo-code fragment representing the quicksort algorithm is shown in Figure 5.2.

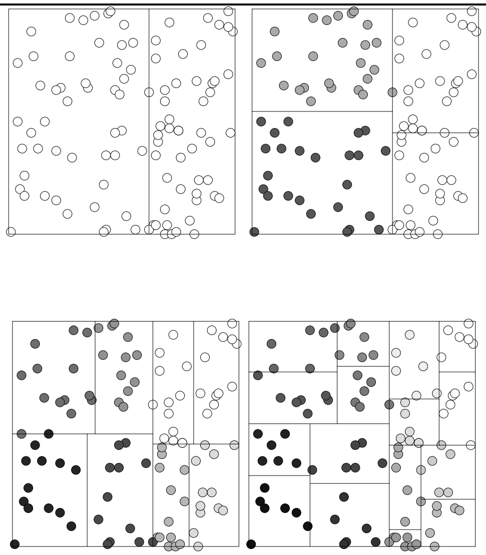


Figure 5.1: Partitioning a random graph with 100 points graph into 2, 4, 8, and 16 parts

For a detailed description and complexity analysis one might consult [55].

5.1.3 Coordinate Bisection

With the help of the sorting algorithm it is now straight forward to write the bisection algorithm. To do so we have to introduce first a few variables.

Since we work in the two dimensional plane it is useful to store the coordinates in two arrays called X and Y. The i-th point has the coordinates (x_i, y_i). Let N denote the number of points placed in the plane. Let *level* denote the number of recursions of the bisectioning, than we divide the points into 2^{level} parts.

The algorithm is shown in Figure 5.3. Note that we changed the sorting routine appropriate. This means that we sort the three arrays X,Y, and names

```
PROCEDURE quicksort (int l, int r, array a)
  i = l;
  j = r;
  x = $a_{\frac{(l+r)}{2}}$
  REPEAT
    WHILE (COMPARE ($a_i$,x) = SMALLER) i=i+1;
    WHILE (COMPARE (x,$a_j$) = SMALLER) j=j-1;
    IF (i≤j) then
      SwapItem ($a_i$, $a_j$);
      i=i+1;
      j=i-1;
    ENDIF
  UNTIL (i>j);
  IF (l<j) then  quicksort(l,j,a);
  IF (i<r) then  quicksort(i,r,a);
END PROCEDURE
```

The routine *COMPARE* returns SMALLER if the value of the first argument is smaller than the second argument.

Figure 5.2: The sequential quicksort algorithm

corresponding to the keys given with the values of X. The integer values a and b specify the range on which the sort should be performed.

5.2 Data Parallel Programming

In case we apply a single instruction to all elements of a data structure simultaneously we use the term *data parallel programming.* A machine capable of performing data parallelism is naturally an SIMD machine. Nevertheless, we are of the opinion that dataparallel programming is more common than often realized. The problem is ow we define an instruction applied on a data structure. An example for a program executed on an MIMD machine which is a dataparallel parallel program are certain parallel genetic algorithms [74, 77, 73]. In this algorithms the data structure is the genetic coding and the instructions are high level instructions like mutation, crossover and selection. In this way

```
PROCEDURE bisection($level$, $a$, $b$, $X$, $Y$)
BEGIN
    $midpoint \leftarrow \frac{a+b}{2}$
    sort $(a, b, X, Y)$
    $coord \leftarrow \frac{x_{midpoint+1} + y_{midpoint}}{2.0}$
    IF level = 1 THEN
        nparts ← nparts + 1
        FOR $i \leftarrow a$ TO $midpoint$ REPEAT
            $partition_i$ = nparts
        END REPEAT
        nparts ← nparts + 1
        FOR $i \leftarrow midpoint + 1$ TO $b$ REPEAT
            $partition_i$ = nparts
        END REPEAT
    END IF
    IF $level > 1$ THEN
        bisection $(level - 1, a, midpoint, Y, X)$
        bisection $(level - 1, midpoint + 1, b, Y, X)$
    END IF
END PROCEDURE
The procedure is initiated with
        bisection $(level, 1, nodes, X, Y)$
```

Figure 5.3: The sequential bisection algorithm

even SPMD algorithms are dataparallel programs. Nevertheless, in this paper we would like to follow the usual understanding of dataparallel programming and their tight connection to predefined instructions an a SIMD machines.

Since the sorting algorithm is used for the bisectioning we try first to parallelize this algorithm. Under the assumption that we can place each data element on a separate processor we can develop a simple sorting strategy often called *odd-even transposition*.

The algorithm is shown in Figure 5.4. First, all odd numbered processors p_i obtain the data element x_{i+1} from the next processor. If the obtained data element is smaller than processor p_i and p_{i+1} exchange there elements stored. Second, all even numbered processors perform this operation. After $\lceil n/2 \rceil$ steps a sorted list is obtained. Figure 5.5 shows an example of sorting 10 numbers.

```
PROCEDURE odd-even-transposition (int l, int r, array a)
 FOR j=1 TO n/2 REPEAT
    FOR $i = 1, 3, ..., 2\lfloor n/2 \rfloor - 1$ DO IN PARALLEL
        IF $a_i > a_i + 1$ THEN
            $a_i \leftrightarrow a_i + 1$
        ENDIF
    END DO IN PARALLEL
    FOR $i = 2, 4, ..., 2\lfloor n/2 \rfloor$ DO IN PARALLEL
        IF $a_i > a_i + 1$ THEN
            $a_i \leftrightarrow a_i + 1$
        ENDIF
    END DO IN PARALLEL
 END REPEAT
END PROCEDURE
```

Figure 5.4: The sequential quicksort algorithm

5.2.1 Radix Sort

The complexity analysis shows that this rather straight forward sorting algorithm does not perform well. Therefore, we introduce here a data parallel sorting algorithm as used by the CM-2 [44].

The CM-2 has the ability to access data in parallel, so that sorting of the data can be avoided often while using ranking. Due to experiments on the CM-2 Hillis and Steele found out that for this particular machine bitonic sort has a very good performance for large sort keys but Radix sort performs better on shorter keys up-to 25-32. Sorting 65536 32 bit numbers takes with both algorithms about 30 milli seconds.

The Figure 5.6 shows the radix sort algorithm. Sorting is performed via enumeration due to significant bits. Since each loop is done over n elements and k is to be assumed much smaller than the number of elements the time complexity for radix sort is $O(n)$.

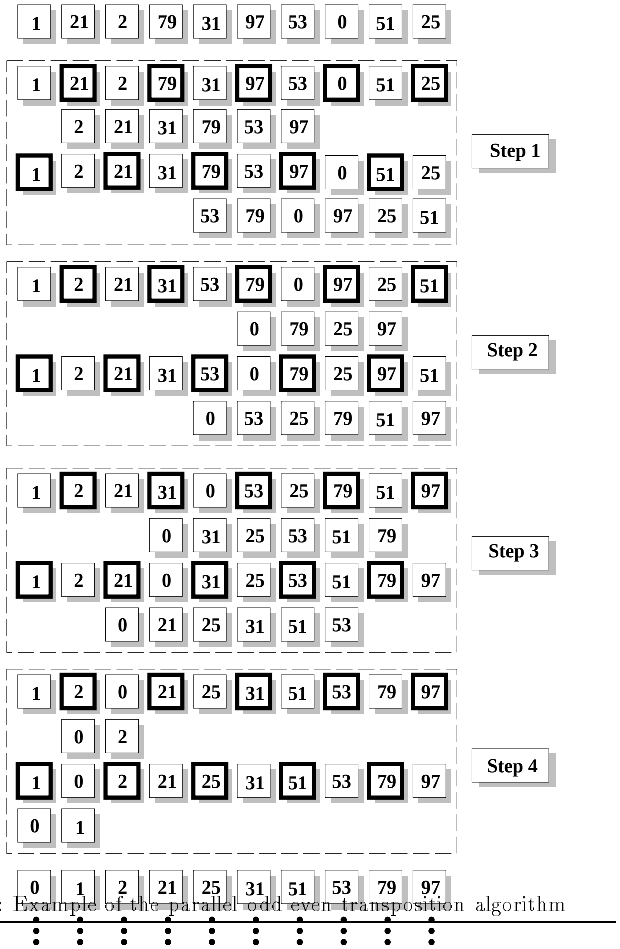


Figure 5.5: Example of the parallel odd even transposition algorithm

```
FOR $j \leftarrow 1$  TO $1 + \lfloor log_2\ maxint \rfloor$ DO
 FORALL $k \in \{1, ..., P\}$ DO
   WHERE $(x_k\ mod\ 2^j) =\ 0$ DO
     use a stable sort array A on digit k
   END WHERE
 END FORALL
END FOR
```

Figure 5.6: A data parallel radix sort algorithm

5.3 Message Passing Programming

5.3.1 Message Passing Sort

The message passing sorting algorithm is dependent on the topology assumed on the processing elements. To be most general only a bidirectional array is assumed as topology for the sorting process. It is important that the sorting process includes only a subset of the computational units so that the other processors might work independently on another task. This enables one to use a parallel machine for example for more than one independent sorting processes as used in the parallel bisection algorithm.

The parallel sorting algorithm itself uses the sequential quicksort in its initial step. Since the quicksort algorithm is outlined earlier the parallel merge sort algorithm is described next.

Parallel Mergesort Algorithm

The parallel sorting algorithm is based on the technique described in [6, 7] under *Merge-Splitting Sort*. This algorithm is based on the Odd-Even Transposition Sort but takes into account that not every element can be stored on a separate processor. This is a very realistic restriction since MIMD machines are build only with a medium number of processing elements.

The example shown in Figure 5.9 illustrates the parallel mergesort algorithm. On the four processors 16 data items are generated in random order. These data items are sorted with the sequential sorting algorithm. The result is shown in

```
    PROGRAM bisection
c       ————- Variables ————————
        INTEGER N, NSegment
        PARAMETER(N=32)
        INTEGER X(N), Y(N), Xnew(N), Ynew(N)
        INTEGER rank(N)
        INTEGER i,j, levels, parts
        LOGICAL segments(N)
c       ————— Determine Home ——————
        X = 0
        Y = 0
        Xnew = 0
        Ynew = 0
        rank = 0
        segments = .FALSE.
c       ————— Initialize ————————
        call CMF_random (X,N)
        call CMF_random (Y,N)
        levels = 2;  parts = 4         ! $parts = 2^{level}$
c       ————— Do main Loop ———————
        NSegment = N/2
        call CoordBisection (N, X, Y, segments, Xnew, Ynew, rank,
      $     NSegment, levels)
    END
c   —————— RankSort —————————
    SUBROUTINE RankSort (N, X, Y, segments, Xnew, Ynew, rank)
        INTEGER X(:), Xnew(:), Y(:), Ynew(:), rank(:)
        INTEGER N
        LOGICAL segments(:)
        call cmf_rank (rank, X, segments, 1, CMF_UPWARD,
      $     CMF_SEGMENT_BIT, .TRUE.)
        Xnew(rank) = X
        Ynew(rank) = Y
    END
```

Figure 5.7: A data parallel FORTRAN program

```
     SUBROUTINE CoordBisection (N, X, Y, segments,
    $     Xnew, Ynew, rankN, Segments, level)
     INTEGER X(:), Xnew(:), Y(:), Ynew(:), rank(:)
     INTEGER N, NSegments,  level
     INTEGER  i
     LOGICAL segments(:)
     DO i=level,1, -2
       IF (level >= 1) THEN
         call RankSort (N, X, Y, segments, Xnew, Ynew, rank)
         FORALL (i=1:N, MOD (i-1, NSegments) == 0)
    $          segments(I) = .TRUE.
         NSegments = NSegments/2
       ELSE
         X = Xnew; Y = Ynew
       ENDIF
       IF (level >= 1) THEN
         call RankSort (N, Ynew, Xnew, segments, Y, X, rank)
         FORALL (i=1:N, MOD (i-1, NSegments) == 0)
    $          segments(I) = .TRUE.
         NSegments = NSegments/2
       ENDIF
     END DO
     END
```

Figure 5.8: A data parallel FORTRAN program

the first row of Figure 5.9.

In the next step all elements of processors with even numbers are send to the right and the ones with odd numbers are send to the left. Each processing element consist now of two data sequences. Its own and the one from its corresponding neighbor. On each even processor the smaller elements of the two data sequences are determined and for the odd ones the larger ones. They replace the data items stored in the processor. This is shown in the Figure 5.9 in the second row.

The following step is similar to the above one with the difference that now each odd processor sends its data to the left and the even ones to the right. This is done with two exemption. The processors on both ends of the linear array do not participate in the exchange process. Than, on each odd processor the

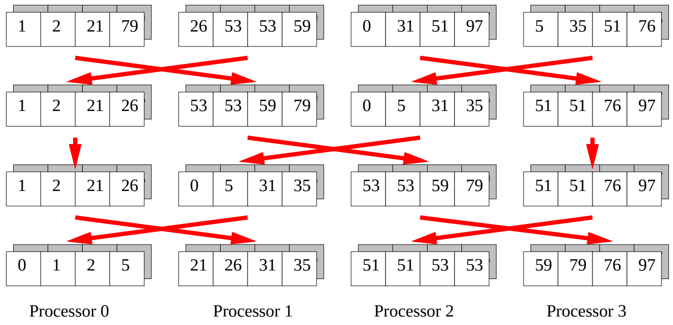


Figure 5.9: Example of the parallel sorting algorithm on 4 processors and 16 data items

smaller elements and on each even the bigger ones are collected.

These two steps are repeated $p/2$ times in order to guaranty the correctness of the sorting algorithm, where p specifies the number of processors.

Complexity Analysis The sorting of n numbers with the sequential quicksort algorithm in each processor is done in $O(\frac{n}{p}\log\frac{n}{p})$ steps. Transferring the data elements from one processor is done in $O(\frac{n}{p})$ steps. A mergesort of two lists requires at most $2\frac{n}{p}$ steps. Thus, the two steps are computed $O(\frac{n}{p})$ time steps. Since they are repeated $p/2$ times, the total running time is

$$t(n,p) = O\left(\frac{n}{p}\log\frac{n}{p}\right) + O(n)$$

In contrast to [6, 7] the following mergesort algorithms are used in the hope to improve the average running time for the mergesort algorithm with the factor 2. The worst case complexity analysis maintains unchanged.

The procedure *mergehigh* mergesorts two lists from the highest to the lowest element till the new list has half the elements from both of the lists.

The procedure *mergelow* mergesorts two lists from the lowest to the highest till the new list has half the elements from both of the lists.

The Figure 5.10 shows the parallel mergesort algorithm in pseudo code.

```
SendRecv (Destination)
 send the own items to destination and receive the neighbor items


PROCEDURE mergesort-parallel
  FOR (t=0; t< ProcessorsInArray; t++)
    IF (even(t)) THEN
      IF    (even(me)) THEN
        SendRecv(right);
        mergelow(dimension);
      ELSE IF (odd(me)) THEN
        SendRecv(left);
        mergehigh(dimension);
      ENDIF
    ELSE
      IF ((me = FirstProcessor) or (me = LastProcessor)) THEN
        no msg exchange
      ELSE
        IF    (even(me)) THEN
          SendRecv(left);
          mergehigh(dimension);
        ELSE IF (odd(me)) THEN
          SendRecv(right);
          mergelow(dimension);
        ENDIF
      ENDIF
    ENDIF
  END FOR
END PROCEDURE
```

Figure 5.10: The parallel mergesort algorithm

5.3.2 Parallel Recursive Bisection

With the help of the above described procedures the recursive bisection algorithm is given in Figure 5.11 with the functionality as described at the beginning of Section 5.1.1 .
The bisection algorithm is initiated with the call:

bisection (0,$p-1$,0,level,$\frac{n}{p}$).

The variable level determines that the input data is partitioned into 2^{level} parts.

```
PROCEDURE bisection (FirstProcessor, LastProcessor,
                     dimension, level, ProcItems)
    quicksort (0, ProcItems-1, items, dimension);
    midpoint = ( LastProcessor - FirstProcessor ) / 2;
    mergesort-parallel ( FirstProcessor, LastProcessor, dimension);
    IF level > 1 THEN
      NextDimension = (dimension + 1) MODULO MaxDimension;
        bisection (FirstProcessor, midpoint,
                   NextDimension,level-1,ProcItems);
        bisection (midpoint+1,   LastProcessor,
                   NextDimension,level-1,ProcItems);
    END IF
END PROCEDURE

Here the quicksort algorithm is extended to the different dimensions
of the input data.
```

Figure 5.11: The parallel mergesort algorithm

5.4 Vector Programming

Using a machine like a CRAY vectorcomputer is for this particular case possible while using the sequential program. The program might be changed in the way that optimized library functions for the sorting algorithm are used. Because the vector computer uses the sequential program as basis we do not describe the specific features further.

Since it is so easy to run sequential programs on a vector computer many scientific programs are written for this class of machines. Nevertheless, in some cases the performance of the machine is badly if the program does not consists of a data structure that computes successively on data elements of an array.

Chapter 6

Performance Taxonomy and Analysis

In order to evaluate the advantegas or disadvantages of a supercomputer it is necessary to introduce some formalism. To compare the running time of an application between sequential and parallel computers the term speed-up is often used.

6.1 Seed-Up

The speed-up for a given problem is defined as the ratio between the response time using a single computer and using a parallel computer with N processors. Let t_{seq} denote the time to solve the problem with a single processor and $t_{par}(P)$ the time to solve the problem with P parallel processors. Than the speed-up is

$$S(P) = \frac{t_{seq}}{t_{par}(P)} \tag{6.1}$$

Sometimes it is not possible to obtain the running time for a problem running on a sequential machine due to limitations of the sequential computer. The time to complete a calculation could be to long or the machine does not provide enough resources, e.g. memory. Therefore, estimates for the sequential running times are used often. This indicates once more the usefulness and the need of

parallel computers for particular applications which could not be tracked by a sequential machine.

6.1.1 Amdahls Law

Amdahls law [8] was one mayor criticism against the use and the development of parallel computers. It says that if α is the proportion of an algorithm which can be performed in serial and $1 - \alpha$ the proportion executed in parallel, then the speed-up achieved with N processors is.

$$S(P) = \lim_{N \to oo} \frac{1}{\alpha + \frac{1-\alpha}{P}} = \frac{1}{\alpha} \tag{6.2}$$

This means no matter how many processors one uses, the speed-up can not exceed α^{-1}. The observation from Amdahl is valid, but one should consider that for making parallel computing useless one uses the assumption that α is large. In contrast for many real life applications we find that α is very small due to the fact that often the parallel algorithm is completely new designed for the parallel computer.
Gustavson et. al. give the explanation why we can achieve for most problems any desired efficiency on any numbers of processors if we scale up the problem size sufficiently[?]. Let

$$t_{seq} = t_\alpha + t_{1-\alpha}^{(N,P)} \tag{6.3}$$

where t_α is the time of the sequential part of the algorithm $t_{1-\alpha}^{(N,P)}$ is the time of the parallelizable algorithm using P processors applied to to problem of size N. Than the speed-up is

$$S(P) = \frac{t_\alpha + t_{1-\alpha}^{(N,1)}}{t_\alpha + t_{1-\alpha}^{(N,P)}} \tag{6.4}$$

Under the assumption that $\frac{\delta t_{1-\alpha}}{\delta N} > 0$ and $\frac{\delta t_{1-\alpha}}{\delta P} < 0$, then any desired efficiency can be reached. For example, assume $t_{1-\alpha}^{(N,P)} = \frac{N^3}{P}$, then:

$$lim_{P \leftarrow oo} S(P) = \frac{t_\alpha + N^3}{t_\alpha + \frac{N^3}{P}} = \frac{Pt_\alpha + PN^3}{Pt_\alpha + N^3} = P \tag{6.5}$$

Thus, the efficiency approaches 1 for large problem sizes.

6.1.2 Linear and Superlinear Speed-Up

Is the speed up directional proportional to the number of processors in the computer used we achieve *linear speed-up*. Due to Amdahls Law linear speed-up cannot be achieved, if the program contains a sequential portion. However, Gustafson's Law states it can be achieved if the problem size is increased as the number of processors is increased. A program is called to be scalable if its performance growth linear with the problem size.

When achieving a speed-up which is greater than the number of processors one reaches *superlinear speed-up.* As we saw superlinear speed-up is theoretically not possible. Neverthele¡ss, in practise it might occur because the algorithm might change while distributing it on a parallel machines. An example for this might be a non deterministic heuristic which may terminate earlier while running it in parallel.

6.2 Efficiency

The concurrent efficiency is defined as

$$\epsilon = \frac{S(P)}{P} \tag{6.6}$$

by combining the equations 6.1 and 6.6 we obtain

$$\epsilon = \frac{t_{seq}}{Pt_{par}(P)} \tag{6.7}$$

The concurrent efficiency is e measure for how well the parallel processors are utilized. The closer the value is to 1.0 the better is the efficiency.

6.2.1 Iso-Efficiency

With the help of the iso-efficiency one can quantify the effect of the size of problem on an algorithm's efficiency.

Let ν be the efficiency, than the iso-efficiency specifies the size of a problem which must be solved on p processors in order to achieve the same efficiency.

6.3 Overhead

Different causes for the reduction of the speed-up and efficiency of a parallel program exist. Those causes include

- Algorithmic Overhead,
- Software Overhead,
- Load Balancing,
- Communication Overhead,
- and External Communication Overhead.

6.3.1 Algorithmic Overhead

While determining the efficiency one compares a sequential and a parallel algorithm solving the same problem. Sometimes the parallel algorithm is difficult to find. In case a parallel efficient algorithm is non existing or can not be used an *Algorithmic Overhead* exists.

6.3.2 Software Overhead

While decomposing an algorithm it might be necessary to introduce a more complex algorithm. This is often the case while parallelizing sequential algorithms on MIMD machines. The programs become more complex in order to ensure data consistency and computational correctness. The overhead introduced by expanding the sequential program is called *Software Overhead.*

6.3.3 Load Balancing

One of the most discussed issues in parallel computing is the load balancing. In order to achieve a high efficiency it is obvious that the computational load between the processors should be distributed evenly.

A problem is load balanced among P processors if the work is evenly distributed among the available processors and the computation is executed at the same time.

The terms mapping and scheduling are tightly connected to load balance. We define these terms as follows: The allocation of processes to a processor is called *mapping*. The allocation of work to processors is called *scheduling*.

6.3.4 Communication Overhead

Communication between processors cost time. This communication is necessary in many algorithms to distribute the data and to inform other processors about data necessary for their computation. This overhead caused by interprocessor communication is called *Communication Overhead*.

6.3.5 External Communication Overhead

A very important issue is the question where the data is obtained on which the calculation is performed. Most parallel programs require external data. These are for example I/O operations to and from disks and external devices such as monitors. Often this devices are shared and an overhead is caused. Therefore, a considerable amount of research is done in parallel I/O these days. The overhead is called *External Communication Overhead*.

6.4 Computation and Communication Time

The total time spend to solve a problem on a parallel machine can be conveniently divided into the computation and calculation time. This has for example also the advantage that on can compare the computation time of a sequential algorithm with the computation time of the parallel algorithm. The communication time can be viewed as overhead necessary to perform the calculation on the parallel machine. We use the abbreviations t_{com} and t_{calc}.
Therefore, we define

t_{calc} = typical time to perform a generic calculation

t_{com} = typical time taken to communicate a word between two processors

t_{ext} = time to communicate a word to send from each processor one word to an external device

In conjunction to t_{com} one finds in literature often the term latency. *Latency* specifies the time taken to service a request which is independent of the size or nature of the operation. The latency of a message passing system is the minimum time to deliver a message.
Furthermore, we define the total communication and computation time

$$T_{calc} = \text{ sum of all generic calculation times } = a(n)t_{calc} \tag{6.8}$$

$$T_{comm} = \text{ sum of all communication times in a program } = b(n)t_{com} \tag{6.9}$$

Let F_C denote the fractional communication overhead

$$F_C = \frac{T_{comm}}{T_{calc}} = \frac{a(n)}{b(n)}\frac{t_{com}}{t_{calc}} = c(n)\frac{t_{com}}{t_{calc}} \tag{6.10}$$

$$\frac{1}{\epsilon} - 1 = F_C$$

The fractional communication overhead is only dependent on the grain size n. Therefore, F_C is independent of the number of processors. Similar to the fractional communication overhead one can define the fractional external I/O overhead.

$$F_E = d(n)\frac{t_{ext}}{t_{calc}} \tag{6.11}$$

For the hypercube designed at Caltech [31] $Tio = Pt_{com}$ so that F_E and F_C differ only in a fractional constant.
F_E^{-1} and F_C^{-1} can be interpreted as the average number of calculations performed per communicated word via internal or external channels. Therefore, it is important

> to do a substantial amount of calculation in ratio to the time spend for the calculation to be efficient.

For some applications the problem occurs that while adding more processors to solve a fixed problem the computation time increases. The point where this occurs first is called *parallel balance point*. An example for a balance point can be found in the parallel LU factorization algorithm shown in [78, 79]. In this example the performance degrades because of the increasing load imbalance and communication overhead.

6.5 Message Passing Environments

The communication between different processing elements is necessary to coordinate the calculation on data distributed over the different processors.

There are many message passing environments available for the different parallel computers. Some of the most common and well known are CMMD, iPSC-NX2, PICL, P4, PVM, Express [4, 33, 19, 15]. These environments consist of a set of functions which enable to write programs using message passing between processing nodes. While some of the libraries are only available for a particular machine others are supported on a variety of machines. The ones especially designed for a particular machine have often better performance then those implemented on many machines. Nevertheless the overhead is small in most cases.

All of them share common characteristics. Beside this basic characteristics they might include considerable extensions. An example for a more powerful environment is Express supporting even graphics capabilities.

Recently, the packages PVM and P4 are used on many machines. Due to its distribution PVM might well become a defacto standard for research and industry. In addition to the available packages there exists the desire to develop an uniform message passing interface system (MPI). A promising step into this direction might be a library combining PVM an P4. This library is know under the name *Chameleon* [69]. Since P4 is supported currently on more machines it might be a way to use this package for developing algorithms for the widest range of massively parallel computers but also heterogeneous networks of workstations.

We feel that a standardization of the interface over many platforms will help to make parallel programming more transparent and less error prone.

All libraries use the model of a virtual concurrent processor. They are a collection of routines supporting the message passing functionality. All systems have

1. a unique identification number of a process
2. and store the total number of participating processors.

The nodes are in connection with each other. As a programmer it is convenient to view the system as fully interconnected which is simulated with software in case the interconnection topology is restricted. We expect that in future this software is located in specific very fast hardware units making the difference between fully interconnection and restricted topologies smaller. Often only a restricted topology is required while mapping a task graph onto the actual machine. In this cases sophisticated mapping strategies may be used to achieve the goal to distribute the load evenly while minimizing the communication overhead.

Many systems underlying a one to one mapping between processes and available processors. This is due to the high cost of swapping necessary while switching from one process to another process execution.

Communication

Many programs are developed with the help of *expected communication*. This means that the participating nodes in the communication coordinate the transmits and receives in an organized way. Consequently, a node can not write to another node unless the destination node is ready for the transmission. One way to implement this *loosely synchronous* communication scheme is by defining blocking communication routines as described in [31].

Another important class of communication routines is know under asynchronous message passing for example used in CMMD. Here the nodes that wish to send or receive data do not block while waiting for the partner node. This has the advantage that a node can send a message and immediately start calculating, enabling the overlap of communication and computation. Using asynchronous message passing is inherently more difficult than synchronous message passing since programming errors might lead easily to deadlocks which are more difficult to detect.

In order to simplify the communication process it is useful to develop some communication routines often used. These routines involve all or a subset of nodes. They are known as collective communication routines. Routines which are useful are

- shift, which moves data between neighboring processors
- broadcast, which sends the same message to the processors
- combine, which executes a function on a data distributed over the processors.

The global *sum* is a good example for a global combine operator. Here the sum of all elements of an array distributed on the processing nodes is calculated and the result is written in a variable available on each node.

Using the basic communication routines one can design libraries with better functionality for a particular problem.

Examples are the routines *gridinit, gridcoord* introduced in [31], libraries for finite differences methods, ScaLAPACK and many others. Essential for the definition of these libraries it the ease they can be used and the scalability of the performance while increasing the number of processors.

6.6 Parallel IO

Many programs are transforming data stored on external devices. This leads to the problem of accessing the data to transform it. In case of parallel programs it is useful to support a concurrent IO subroutine library. The CUBIX library functions provide a way to do this.

We distinguish two different modes. The first mode is called *singular mode*. In this mode each node of the ensemble must concurrently execute an identical IO operation (the routines are called in loose synchronization). The IO operation is executed when all nodes have arrived at the same rondezvous point. In the second mode, *multiple IO mode*, distinct data bound to different processor can be accessed. It is useful to be able to switch between the separate modes. An example can be found in [31] on page 114.

Since IO is one bottleneck in parallel programs research is conducted in this area recently. One study [18] shows that it is from advantage to provide a two step library for IO operations enabling the optimization for particular machines more easily. The study shows that for different distributions as used in HPF the routines provided by the vendor (Intel Delta) lack of a high performance. In order to speed up the computation the authors suggest to load in the data with the fastest IO routine possible and redistribute the data as necessary in a second mapping step. With this strategy for IO primitives difficult data distributions like the (block,block) distributions are possible.

Chapter 7

Applications

7.1 Numerical Integration

The integration of a continuous function is an embarrassingly parallel problem. Therefore, it can be easily implemented in both a message passing and a data parallel program. To determine the integral

$$\int_a^b f(x)dx$$

the interval inbetween a and b can be divided into equally large parts.
For each part a representant of the function has to be found. This representant is than multiplied by the size of the subinterval. Adding up all those areas in the interval between a and b results to an numerical approximation of the function. Increasing the number of subintervals inbetween a and b increases the accuracy of the computation.
A sequential algorithm can be formulated as follows:

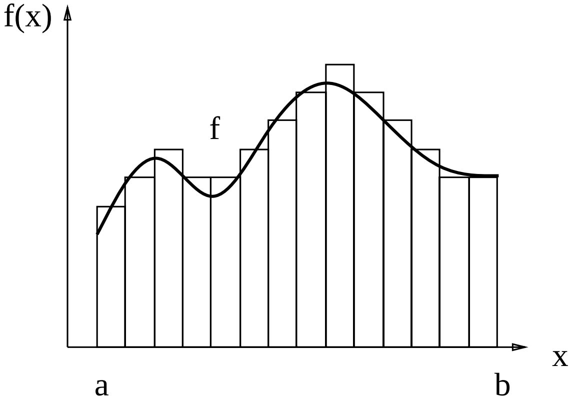


Figure 7.1: Integration of a function

```
$n$ = number of subintervals.
$\Delta x = (b - a) / n$
sum = 0.0
$x = a$
while ($x < b$)
  $y = f(x)$
  sum = sum + $y$
  $x \leftarrow x + \Delta x$
end
```

Since the single computational steps are independent from each other the algorithm can be easily parallelized:

```
$n$ = number of subintervals.
$\Delta\ x\ =\ (b\ -\ a)\ /\ n$
forall $i\ \in\ 1,\ ...,\ n$ do in parallel
  $x_i\ =\ \Delta\ x\ *\ i$
  $y_i\ =\ f(x_i)$
end
forall $i\ \in\ 1,\ ...,\ n$ do
   sum = sum + $y_i$
end
```

The algorithm consists of two parts. The first part computes the function value for each subinterval and the second part computes the sum.

7.1.1 Dataparallel Program

On the connection machine exists an efficient implementation of the sum operation. It is called SUM and sums up all the values of a vector distributed over different processing elements.

```
$n$ = number of subintervals.
$\Delta\ x\ =\ (b\ -\ a)\ /\ n$
forall $i\ \in\ 1,\ ...,\ n$
  $x_i\ =\ \Delta\ x\ *\ i$
  $y_i\ =\ f(x_i)$
end
sum = SUM (y:1:n)
```

7.1.2 Message Passing Program

The message passing program is slightly more complicated since on each processor are assumed to be more than one data item. Let p denote the number of processors and n/p are the number of integration points mapped on one processor, than the algorithm for the integration looks like:

Dataparallel Program

```
$p$ = number of total processors
$id$ = identification number of own processor
$n$ = number of subintervals.
$\Delta\ x\ =\ (b\ -\ a)\ /\ n$
forall $i\ \in\ 1,\ ...,\ n$
  $x_i\ =\ \Delta\ x\ *\ i$ + $id\ \frac{n}{p}$
  sum  = sum + $f(x_i)$
end
sum = GlobalSum (sum)
```

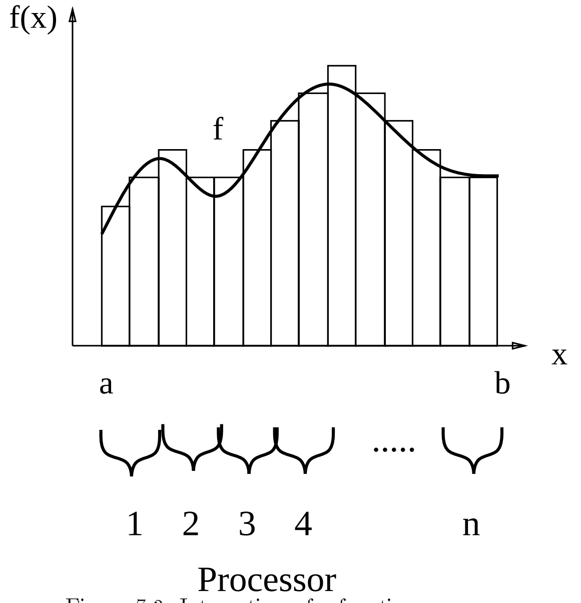


Figure 7.2: Integration of a function

Determining the GlobalSum of elements (see section ??) is done in $O(logn)$ time on a variety of architectures including the CM5 and the Hypercube by using a tree structure for the summation process.

Message Passing Program

Therefore, we compare the complexity of the algorithm.

Algorithm	Time	Space	Cost
sequential	$O(n)$	$O(1)$	$O(n)$
parallel(n=p)	$O(logn)$	$O(n)$	$O(nlogn)$
msg parallel	$O(logp) + O(\frac{n}{p})$	$O(p)$	$O(plogp) + O(n)$

In the table the cost is defined as space times time cost. An optimal parallel architecture would have naturally the structure of a tree to allow the efficient summation of the subintervals.

7.2 Long Range Interactions

A problem is in the class of Long Range Computations if all nontrivial computation must be performed between all pairs of a data base. Examples for long range computations are gravitational n-body problems, products of two long polynomials, molecular dynamics calculations, and others.
The simple way to calculate the result is to loop over all element pairs of the database. Assume that D is the domain or the database and $p_i \in D$ denotes the i^{th} point in the database Let $f(p_i, p_j)$ denote the calculation performed between the points i and j, than the following loop describes the generic calculation.

```
do i=1,n
  do j=1,n
      $f(p_i, p_j)$
  end do
end do
```

Clearly, the complexity of the calculation is $O(n^2)$. To calculate this function in parallel one can use a simple pipelined approach for solving the problem. Assume we have m processors and N is the total number of particles. For simplicity we assume that m divides N without rest. At the beginning each processor holds two identical arrays of N/m particles. Than in each processor

the calculations between each particle in this processor are calculated. Now the particles of the second list are send to the neighboring processor in left direction.
This step is repeated $m - 1$ times.
Looking at the overhead:

$$f \sim \frac{1}{m} \frac{t_{sendreceive}}{t_f}$$

one can clearly see that the algorithm will perform efficiently as long as the number of particles stored in one processor are large enough.

Optimization

In case we know some properties of the function to be calculated the performance of the algorithm can be improved. For example, in case of n-body calculations we know that the potential between the same point is 0 and the function is symmetrically. Therefore, we only need to calculate $n^2 - n$ points making the algorithm almost twice as fast. This is due to the reduction of the calculation by the factor of more than half but keeping the communication time constant.
The algorithm is easily extendable in case N is not divided by m without rest. In order to maintain load balance we just keep track that every processor contains approximately the same number of particles.

7.3 Short Range Interactions

The N-body calculation introduced above is computationally very intense. One can reduce the computational effort while making use of the physical structure of the problem and introduce short range interactions.
In this section we concentrate on the problem where an update of the particles, e.g. a positional change has to be calculated. This algorithm is characterized by it's need predictable communication pattern. That means we know that a communication phase follows the computation phase but we do not know which data has to be transmitted to which processor at compile time.
A simple but not necessarily best data decomposition for this problem is to divide the geographical regions evenly over the processors. Better methods are

for example the barnes hut method and the introduction of a particle tree.

Here we introduce the scheme of the geometrical hashing. We assume that a particle can only move a maximum distance which is smaller than the area covered by one processor. This has the advantage that only nearest neighbor processor communication occurs. Due to the advanced operation system of most of the parallel machines is is easy to introduce diagonal moves in the communication pattern. Most parallel machines allow an arbitrary point to point communication. With this advanced operation systems the restriction of the minimum distance a particle travels can be overcome but results in higher communication overhead on most of the available machines.

The way one can update the particle movement is to store a list with 8 queues in the 2d case representing the 8 directions. In this queues one inserts the elements which have to be moved to another geographical block and therefore processor. After a particular time the particles are simply send over to the processor where it should be placed. With the help of asynchronous communication this could be achieved quite easily. In the receiving processor we interrupt the calculation if a message arrives an incoming channel. This message/particle is than included in the list of particles of this particular processor. In order not to destroy the actual calculation the polling of the incoming messages is done after a calculation is completed. Is no message available the processor can continue with its calculation.

Geographical hashing comes in place when the calculation is performed on only part of the articles rather than the complete database of particles reducing communication and computation effort drastically. Assume the weight of a particle taking part of a calculation is related to its distance. Assume that local interactions are far more important than long range interactions. In this case it is important to consider only particles in the local neighborhood smaller than a cutoff distance. Under the assumption that the cutoff distance is smaller than the geographical region a processor stores only particles in neighboring processors have to be considered reducing the calculation and communication cost. The communication for border elements can be done in a similar way to the finite difference and finite element method.

7.3.1 Irregular Problems

In case the distribution of particles or unevenly distributed communication over the processors the algorithm above would not perform well because of a high load imbalance. The Wator problem introduce in [31] is a good example of this class of problems. A geographically domain decomposition seems to be not suitable. Nevertheless one could introduce smaller geographically blocks as shown in the next picture where the numbers represent the distinct processors.

```
2121212        instead of    11112222
34343434                      11112222
12121212                      33334444
34343434                      33334444
```

This is a fairly easy domain decomposition, known as scattered decomposition. It works for the WaToR problem. For the N-body problem the situation is not so simple and it is better to distribute for example the linked list of neighbors in a hash oct tree fashion as shown by S. Warren in SC93. The idea here is first to generate a hash tree and than distribute the particles evenly on a ring of processors using the peano-hilbert ordering over the domain.

7.4 The Simulated Annealing Approach

Simulated Annealing (SA) is an optimization technique simulating a stochastically controlled cooling process[50, 27, 53, 54]. The principles of simulated annealing are based on statistical mechanics where interactions between a large number of elements are studied. One fundamental question in statistical mechanics is to study the behavior of such systems at low temperatures. To generate such states the system is cooled down slowly.

As the name indicates *Simulated Annealing* simulates such an annealing process. The process is started at a random state s. Now the aim is to find a configuration t with lower energy obtained by a random disturbance of the current state s. The probability to get from one system configuration into the next can be described by a probability matrix (a_{st}) where a_{st} specifies the probability to get from configuration s to a *neighbor* configuration t. This probability matrix is chosen symmetrically. Is

$$E(t) - E(s) < 0,$$

the new configuration is accepted, since it has a lower energy. Is the energy of the new configuration t higher, than it is accepted with the probability proportional to

$$e^{\frac{E(t)-E(s)}{T}}.$$

The acceptance of the configuration with the higher energy is necessary to allow jumps out of local minimal configurations. This process is repeated as long as an improvement is likely. The choice of the temperature parameter is given by a cooling plan such that

$$T_1 \geq T_2 \geq ..., \text{ such that } \lim_{k \to \infty} T_k = 0$$

This leads to the generic simulated annealing algorithm shown in Figure 7.3.

7.4.1 SA and GPP

As example we use the Graph Partitioning Problem as introduced in [75, 51, 52]. For the graph partitioning problem one can find the following analogies between the physical system:

physical system	SA-GPP
state	feasible partition
energy	cost of the solution
ground state	optimal solution

Graph Partitioning

```
PROC Generic Simulated Annealing
   BEGIN SEQUENTIAL
      chose the cooling strategy $T_1$, $T_2$, ...
      generate a starting solution $s$
      $k \leftarrow 1$
      REPEAT
         generate a new solution $t$ in the
           neighborhood of $s$
         $\Delta E \leftarrow E(t) - E(s)$
         IF $\Delta E < 0$ THEN
            $s \leftarrow t$
         ELSEIF $e^{-\frac{\Delta E}{T_k}}$ < random [0,1] THEN
            $s \leftarrow t$
         END IF
         $k \leftarrow k+1$
      UNTIL $T_k \leq T_{min}$ or time limit reached
   END SEQUENTIAL
END PROC
```

Figure 7.3: The generic simulated annealing algorithm

The uniform graph partitioning problem (GPP) is a fundamental combinatorial optimization problem which has applications in many areas of computer science (e.g., design of electrical circuits, mapping) [51, 65]. The term *graph partitioning problem* is used in literature for different problems. Following the paper [50] and the notation in [65] the graph partitioning problem can be formulated mathematically as follows:

Let $G = (V, E)$ be an undirected graph, where $V = \{v_1, v_2, ..., v_n\}$ is the set of n nodes, $E \subseteq V \times V$ is the set of edges between the nodes. The graph partitioning problem is to divide the graph into two disjoint subsets of nodes V_1 and V_2, such that the number of edges between the nodes in the different subsets is minimal, and the sizes of the subsets are nearly equal. The subsets are called *partitions*, and the set of edges between the partitions is called a *cut*.

Figure 7.4 shows a simple graph and a possible partition of this graph. The cost of this solution is 3 assuming that each edge has the weight 1.
For some algorithms it is from advantage not to maintain the strict constrained of the equal partition size. This can be done by extending the cost function with an imbalance term. Than the *cost* of a partition is defined to be
For the GPP we can define the following cost function:

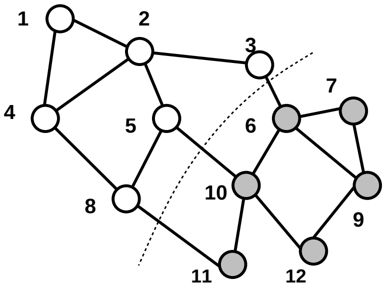


Figure 7.4: Example of a partition

$$c(V_1, V_2) = |\{\{u, v\} \in E : u \in V_1 \text{ and } v \in V_2\}| + \alpha(|V_1| - |V_2|)^2$$

where $|A|$ is the number of elements in the set A and α controls the importance of the imbalance of the solution. The higher α the more important is the equal balance of the partitions in the cost function.

Using the cost function as given in equation (7.4.1) one can obtain solutions to problem instances by an annealing process. The definition of the probability matrix (a_{st}) can be chosen as follows:

- A partition t is neighbor to a partition s iff s can be obtained by moving one node of a subset to the other.
- The probability of the neighbor states to a partition s are the same.

7.4.2 Parameter scheduling for the Simulated annealing algorithm

Instead of using the generic simulated annealing algorithm we use the algorithm displayed in Figure 7.5. This is motivated by the following reasons:

1. It is easier to define a cooling scheme.

2. Many studies published use this scheme which makes comparison more easy.

```
PROC Simulated Annealing
  BEGIN SEQUENTIAL
    $T \leftarrow T_0$
    generate a starting solution $s$
    WHILE (NOT frozen) DO
      DO $i \leftarrow 1, L*$ Number of Nodes
        generate a new solution $t$ in the
          neighborhood of $s$
        (swap one node to the other part)
        $\Delta E \leftarrow E(t) - E(s)$
        IF $\Delta E < 0$ THEN
          $s \leftarrow t$
        ELSEIF $e^{-\frac{\Delta E}{T_k}} <$ random $[0,1]$ THEN
          $s \leftarrow t$
        END IF
        $time \leftarrow time + 1$
      END DO
      $T \leftarrow k * T$
    END WHILE
  END SEQUENTIAL
END PROC
```

Figure 7.5: Simulated Annealing Algorithm used for the Experiments

The simulated annealing algorithm 7.5 is controlled by the parameters $L, k, alpha$. The variation of this parameter has a large influence on the solution quality as shown in the paper by Johnson. We were able to reproduce the results and show some of them in [75]

Acceptance Frequency The acceptance frequency specifies how frequent a change in the solution during a timeinterval of the annealing process occurred. The frequency is calculated over all trials in the inner loop of the interval length L.

This acceptance frequency is important since it helps to estimate when an improvement of the solution will be unlikely while continuing the annealing process. We decided to terminate the algorithm (the solution is frozen) when the acceptance probability drops under 0.00025 percent.

Imbalance Factor The imbalance factor has also an huge influence in the acceptance rate. Is the value to large the process stops to early in a bad solution is it to small we fall in a solution which is very imbalanced and often unwanted. In many studies the value of 0.05 is chosen and we could reproduce with this value very good results.

Temperature The initial temperature is very critical for an efficient simulated annealing run. While choosing the temperature to high no improvement of the solution occurs during the first annealing steps (Figure ??). Is the temperature chosen to low the cooling process terminates to quickly and does not spend time in intermediate solutions. Therefore, the solution space is not explored in depth. For our results we obtained very good results with a temperature of 0.5. The acceptance frequency at this temperature for the given problem instance is about 0.80. Johnson et. al. proposed to reduce the temperature such that the acceptance frequency drops to 0.4. In this case we reduced the running time about 1/3, while keeping the quality of the solutions found.
In case of high temperatures most of the transpositions are accepted. The smaller the temperature get the fewer transpositions are accepted. At temperature 0 only transpositions with positive ΔE are accepted. This effect is shown in Figure ??. A common way to specify the cooling strategy to chose a cooling rate such that

$$T_{k+1} = rT_k \ .$$

With an $r = 0.95$ we obtained very good results.
The following parameters lead to very good results: $T_0 = 0.06, \alpha = 0.05, k = 0.95, L = 16$. Overall the following results for the GPP and Simulated annealing are valid (see above and [50]).

- Long annealing must be used to get the best results.
- It is not necessary to spend a long time at high temperatures.

- A geometric cooling strategy is sufficient
- The variation of the solutions can be large even with long runs.
- The parameter setting is dependent on the problem instance.
- Small neighborhood sizes improve the running time

7.4.3 Parallelization

The simulated annealing algorithm is somewhat difficult to parallelize due to occurring conflicts while updating the costfunction in parallel.
One way to do this is to consider swaps of points in parallel. Therefore, a possible algorithm could be to divide the points onto different processors.
Assume we have m processors and n points in the graph. At the beginning of the algorithm we distribute the points randomly over the processors. Aim is to divide this points between the k processors.
To make the annealing step work we have to store for each point not only the neighboring node but also the processor in which this neighboring node is stored. Now in each processor a point can be selected in parallel to be switched between the processors using the annealing update rule. It is clear that it could come to conflicts which have to be resolved in case two processors request the same node for switching. In order to resolve this conflict the update is done in multiple stages.

1. Each processor selects a random point located in the processor.
2. A random point outside the processor is selected.
3. requests to the processor are send on which the neighbors are located.
4. The request is granted if no conflicts occur. In case of a conflict the node is send to a requesting processor selected randomly.
5. update the costs

This algorithm has two disadvantages.

1. A parallel move can cause moves with contradictory gain win as introduced in Dally [21]. Dally also gives a way how to avoid this.

2. requests to the same processor might be possible causing a high volume of messages to one processor. This could be avoided by introducing around robin processor selecting scheme preferring communication between processors which contain neighbors.

It is out of the scope of this paper to go too much into details. We refer here to the many papers available on parallel simulated annealing algorithms and to [31] where an algorithm for the TSP on a Hypercube architecture is introduced. Besides parallel implementations of the Simulated Annealing [66, 12, 14, 39, 49, 80] algorithm other parallel implementations exists for the Graph partitioning problem, for example [62, 35, 36, 22]

7.5 Matrix Algorithms

7.5.1 Matrix Multiplication

Many scientific problems are based on matrix operations [38, 64, 67]. One very important operation is the matrix multiplication. We introduce here a parallel algorithm for the matrix multiplication. The algorithm introduced here will need more data movement than the other algorithms described in this report.
First, the matrix is decomposed in parts which will be distributed on the different processors. For simplicity we assume that the number of processors is p and that the number of rows and columns are equal. One way to decompose this problem is to choose rectangular decompositions as shown in Figure 7.6. The algorithm consists of four steps. Assume n is the number of columns and m x m are the number of processors. The processors are arranged in a grid or torus.

1. Communicate the diagonal subblocks to all processors in horizontal direction.
2. The subblocks from the last step are multiplied with the appropriate blocks of B stored at this processor and added to the matrix C.
3. The subblocks B are shifted up in the processor grid.
4. The subblocks of A which are to the right of the subblocks A transferred in the first step are broadcasted in the rows. Goto Step 2.

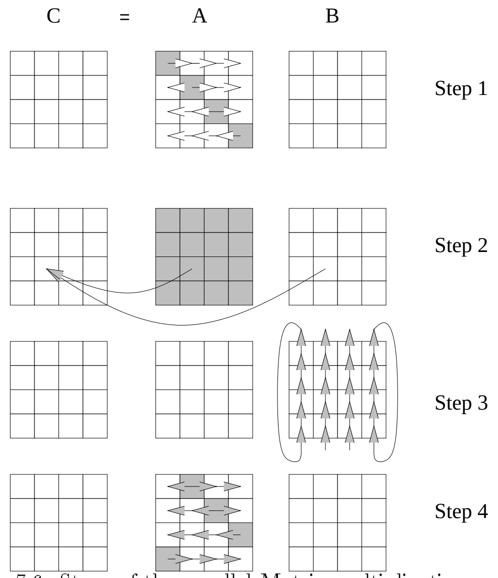


Figure 7.6: Steps of the parallel Matrix multiplication

This is done M times. At the end of the calculation the Matrix C is computed.

Complexity Analysis

The complexity analysis is naturally dependent on the machine used because the broadcast routine might be of different order. We assume in the current analysis that a machine is used with point to point communication and a network topology of a torus.
The time for the broadcast in one row would be of the order

$$(N/M)^2 t_{comm} * (M-1)$$

The time for performing the calculation on a subblock is

$$2(N/M)^2 t_{calc}$$

The time for moving the B subblock is

$$(N/M)^2 * t_{comm}$$

Adding up the three terms gives the total time for one time step. This step is repeated M-1 times.

7.6 LU factorization

Solutions of a system of linear equations are required in many scientific applications[32, 48]. Consider the solution of a dense system of linear equations, $A\vec{x} = \vec{b}$, where A is an n-by-n matrix, $\vec{b}$ is a vector of dimension n and $\vec{x}$ is the solution vector of dimension n. One method of solving this problem is to proceed by first factorizing A into a unit lower triangular matrix L and an upper triangular matrix U, *i.e.*, $A = LU$, and then solving for $\vec{y}$ and $\vec{x}$ in two consecutive substitution steps: $L\vec{y} = \vec{b}$ and $U\vec{x} = \vec{y}$.

7.6.1 Message Passing Algorithm

Most of the CPU time is spent in factorizing the matrix because:

- the computational effort to factorize the matrix is higher than for the two substitution steps.
- most factorization programs result in more memory accesses than floating point operations. This cause the processor to be idle during the time data is transferred from the memory for the computation.

The first observation motivates *why* it is desirable to build a fast LU factorization algorithm. The second observation shows *where* optimization can be successful: It is worthwhile to optimize a factorization algorithm so that it makes efficient use of the way data is transferred to the computational unit.
The memory access time can be decreased if the usage of specific data can be predicted, so that the data can be transferred into a faster part of the hierarchy before it is actually referenced. Since most memory organizations fetch a block of data instead of one datum at a time, it is best to formulate the algorithms in such a way that data elements used in consecutive computation steps are stored in contiguous addresses of the memory. Hence they are fetched in a block requiring fewer memory accesses. Figure 4.2 shows how data (a matrix) is stored in a memory using the column oriented programming language Fortran. Algorithms which update a block of contiguous vectors instead of only one data element at a time are known as *blocked algorithms*. This way the work is done *locally* on a block of data. Data locality is one of the fundamental problems in parallel computing and has a great influence on the performance on

such machines. The use of blocked algorithms is one of the most efficient ways to improve the performance of numerical algorithms on distributed memory machines[78, 60, 61, 78].

Parallel Computational Model

To define a parallel algorithm for solving a system of linear equations it is necessary to define the computational model on which the implementation is based. A study on shared memory MIMD machines using blocked based algorithms for LU factorization can be found in [61]. The results presented in this paper concentrate on distributed memory MIMD machines. These machines have a natural bound on the number of available processors. At a time, each processor can execute different instructions in parallel on different data. Message passing allows interprocessor communication. In order to incorporate a wide variety of architectures, the communication relation between the processors is based on a unidirectional ring.

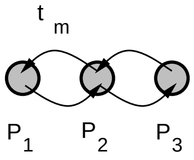


Figure 7.7: Parallel computing modell

Now it is clear that an efficient implementation has to find the trade off between communication time and computation time. If the algorithm sends too many messages and computes too less, the communication time dominates the computation time. Furthermore, we assume that the memory capacity of each processing element is restricted.

Basic Linear Algebra Subprograms

Usage of vector and matrix operations are common in many applications which are not provided by a programming language like Fortran77. Hence, it is desirable to have a library supporting such a class of routines. The **B**asic **L**inear **A**lgebra **S**ubprograms (BLAS) provide such routines [24]. Different levels of BLAS are distinguished by the *amount of data* used for an operation and its *arithmetic complexity*. Computations on vectors of order n can be found in level 1 BLAS. Level 2 BLAS provides matrix-vector computations of order n^2, and level 3 BLAS provides matrix-matrix computations of order n^3.

Looking at the computational effort of the BLAS routines it is clear that the ratio between floating point operations and memory accesses for level 1 and 2 BLAS is not as good as for level 3 BLAS routines. Therefore, it is obvious that the strategy is to maximize the use of level 3 BLAS. The presented algorithms use some subroutines from BLAS of level 3, like GEMM for the matrix multiplication and TRSM for solving a triangular system.

Noblock algorithm

Now, a necessary basis has been established to formulate the factorization algorithms. A fast noblock factorization algorithm forms the building block for the blocked algorithms. The noblock algorithms are distinguished by the order of loops in which the factorization is done. The suitable loop orders for the column oriented FORTRAN are jik, kji, and jki. For example, the abbreviation jik points out that j is the loop index for the outermost loop and k for the inner most loop [25].

```
do ———————
  do ———————      where $i, j, k$ are the loop indices
    do ———————
      $a_{ij} \leftarrow a_{ij} - \frac{a_{ik} * a_{kj}}{a_{kk}}$
    end do
  end do
end do
```

Figure 7.8: Gaussian Elimination Algorithm

Since the number of memory touches for the kji noblock algorithm is twice as high as for the jki-noblock and the jik-noblock algorithm, the running time for this algorithm is slower. Experiments show that the *jik* noblock algorithm performs better than the two others.

jik-Noblock Algorithm

Before the algorithm is described in detail it is useful to visualize the data dependencies of the $n \times n$ matrix elements between the computational steps (Figure 7.9) of the *jik*-noblock algorithm. Data dependencies are expressed by the height of the matrix element. If a datum is higher than another then this datum has to be calculated first. This scheme is used throughout the paper.
In Figure 7.9 the state of the algorithm is shown at time step j. Following the data dependencies first the vector $l^{(j)}$ is updated using $A^{(j)}$ and the vector $u_r^{(j)}$; next the element of l_1^j is computed and $u^{(j)}$ is updated using $U^{(j)}$ and $l_r^{(j)}$.
Therefore, at time step j the *jik*-noblock algorithm updates one *column* of L and one *row* of U. This noblock algorithm is also also known as Crout's Algorithm.

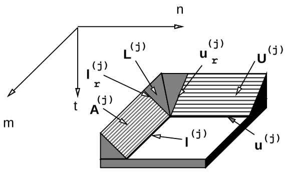


Figure 7.9: *jik* noblock

Let $l^{(j)}$ represent the j^{th} column vector of the matrix L and $u^{(j)}$ the j^{th} row vector of the matrix U beginning at a_{jj}. The matrix $A^{(j)}$ specifies a submatrix of A which includes all elements from the first column to the column $j-1$ and from the rows $j+1$ to n. The matrix $U^{(j)}$ specifies a submatrix of A which includes all elements from the first row to the row $j-1$ and from the columns $j+1$ to n.

```
0. Initialization: $j \leftarrow 1$
1. Update l(j): $l^{(j)} \leftarrow l^{(j)} - A^{(j)}u^{(j)}$
2. Select pivot and exchange:
   $p \leftarrow j + \min\left\{ j \middle| |l_j| = \max_{1 \leq i \leq m-j+1} \{|l_i^{(j)}|\} \right\} - 1$
```

```
   Exchange row j and row p
3. Scaling: $l_i^{(j)} \leftarrow l_i^{(j)}/a_{p,j} \quad \forall i \in [1, m-j+1]$
   IF $j = n$ THEN stop
4. Compute row u(j): $u^{(j)} \leftarrow u^{(j)} - U^{(j)}l_r^{(j)}$
5. Iterate: $j \leftarrow j + 1$
   GOTO Step 1.
```

The detailed description of the other sequential blocked algorithms using BLAS can be found in [61].

Parallel Blocked Algorithms

Rewriting the LU decomposition as a blocked algorithm with properly defined block sizes helps to limit the trade off between the communication time and the computation time. To understand the parallel blocked factorization algorithms the corresponding sequential blocked algorithms are also introduced. This way one is able to observe the data dependencies inherent in the algorithms. The parameter β specifies the block size which is the number of column vectors used by the noblock algorithm for factorization in each iteration of the blocked algorithm. As shown in [78] the parallel SAXPY algorithm is the most general one for our purposes.

Parallel Blocked *kji*-SAXPY

In the j^{th} step of the *kji*-SAXPY algorithm, one block column of L and one block row of U are computed and the corresponding transformations are applied to the remaining submatrix. The basic steps involved in the j^{th} iteration are shown in Figure 7.10 (left). Of the three algorithms presented, this algorithm is best suited for distributed memory MIMD architectures. The reason for this is the data dependencies involved in the steps above (shown in Figure 7.10.) In the j^{th} iteration, the j^{th} block depends only on the $j-1^{th}$ factorized block. Hence, each node has to store only the last factorized block. As a result the memory limitations encountered in the parallel SDOT and GAXPY algorithms do not exist here. Furthermore, the amount of work involved in updating and computing the block row, i.e. the time spent in Steps 2 and 3 (rest of the processors) is comparable to the time required to update and factorize the subdiagonal block (current processor). Hence, the pipelined version of this algorithm produces a significant improvement in performance. The pseudo-code for the pipelined version of the algorithm is given in Algorithm 7.11. Figure 7.10 (right) shows the layout of the matrix onto the processor along with the operations in the second iteration. The activities of each of the processors in the pipelined algorithm are shown in Figure 7.12 for a three processor system.

Results

The presented study was conducted on a 32 node Intel iPSC/860 Hypercube. Each i860 node has an 8 KByte cache and 8 MBytes of main memory. The clock speed is 40 MHz, and each node has a theoretical peak performance of 80 MFLOPS for single precision. Communication is supported by direct-connect modules present at each node.

PICL [33] (Portable Instrumented Communication Library) was used in the presented implementations and provided a simple and portable communication structure. Some of the algorithms used the Basic Linear Algebra Communication Subprogram library for data movement between the processors (BLACS)[11]. BLACS is a part of the effort to implement LAPACK on distributed memory MIMD architectures. The matrices used in the experiments were dense matrices where each matrix element was a randomly generated real number between 0 and 1.

In order to compare the parallel implementations of the three factorization algorithms we first compared the performance achieved for a constant matrix size with different block sizes and on different numbers of processors. The parallel GAXPY algorithm performs better than the other two algorithms for a smaller number of processors. Maximal performance is achieved by the parallel GAXPY algorithm on 16 processors with a block size of 8. The performance of the SAXPY algorithm is close to the GAXPY algorithm and achieves the best performance for block sizes in the range 8 ± 4. The SDOT algorithm achieves very little improvement over the noblock algorithms because of its data dependencies as described earlier.

The performance obtained for very small block sizes is low for all three algorithms. Although small block sizes provide better load balancing, this advantage

is offset by the increased communication overheads between the processors and the decrease in the amount of computation at each processor. Very large block sizes on the other hand, lead to low communication overheads but poor load balancing. This leads to a fall in performance for the parallel GAXPY and SAXPY algorithms as the block size increases. This observation however, does not hold for the parallel SDOT algorithm which shows a slight improvement in performance for very large block sizes. The reason for this is that large block sizes imply smaller number of reshapes which means a smaller penalty is paid by the SDOT algorithm.

An absolute peak performance of about 26 MFLOPS/Processor is achieved by the parallel SAXPY algorithm for a 1536×1536 matrix with β equal to 12 and on 4 processors. It shows that the optimal number of processors is problem dependent.

Conclusion

This section describes methods for blocked LU factorization on distributed memory MIMD architectures. These methods are also applicable on shared memory parallel vector computers [61]. Our numerical results and performance comparisons show the following:

- The data dependencies inherent in the parallel SAXPY algorithm are most suited for distributed memory MIMD architectures. No reshaping of the matrix is necessary since only a small portion of the factorized matrix has to be stored in each processor.

A further observation from our experimentation with LU factorization is that the best performance is achieved at block sizes where the computation at each node outweighs the tradeoff between high load balancing (small block sizes) and low communication overhead (large block sizes). This optimal block size is dependent on the algorithm used, the size of the matrix and the number of processors available [78].

Figure 7.12: Spacetime diagram for the pipelined kji-SAXPY algorithm

7.6.2 Data Parallel Algorithm

Gauß-Jordan Algorithm

The well known Gauß-Jordan algorithm is a straight forward method to find the solution of a system of linear equations. Multiples of the equations are added in such a way that the resulting system has the form of a triangular:
Let A be nonsingular and $a_{i,n+1} = b_i$.

```
do j=1,n
   do i=1,m
      do k=j,n+1
         if ($i \neq j$) then
            $a_{ik} = a_{ik} - (a_{ij}/a_{jj})a_{jk}$
         end if
      end do
   end do
end do
do i=1,m
   $x_i = a_{i,n+1}/a_{ii}$
end do
```

On a SIMD computer like the Connection Machine this algorithm can be parallelized easily by executing the loops for i and k in parallel.

```
do j=1,n
   do i=1,m; k=j,n+1 in parallel
      if ($i \neq j$) then
         $a_{ik} = a_{ik} - (a_{ij}/a_{jj})a_{jk}$
      end if
   end do in parallel
end do
do i=1,m in parallel
   $x_i = a_{i,n+1}/a_{ii}$
end do in parallel
```

Complexity

The complexity of the sequential algorithm is $O(n^3)$. In the parallel algorithm the inner loops for i and k can be parallelized since no data-dependency exists. Therefore, n steps are necessary to compute the outermost. To compute the result an additional step is needed to perform the last parallel loop. Therefore, O(n) time steps are needed to perform the Gauß-Jordan algorithm on $n^2 + n$ processors. Since $t(n) = O(n)$ and $p(n) = O(n^2)$ the cost of the parallel algorithm is $c(n) = t(n)p(n) = O(n^3)$.

Algorithm	Processors $p(n)$	Time $t(n)$	Cost $c(n)$
Gauß(seq.)	1	$O(n^3)$	$O(n^3)$
Gauß(par.)	$O(n^2)$	$O(n)$	$O(n^3)$

In the table the cost is defined as space times time cost.

Gaussian Elimination on the connection machine

On the connection machine the Gaussian elimination can be easily formulated using the SPREAD command. The SPREAD command has the following parameters:

SPREAD (source, dimension, ncopies)

The command SPREAD takes a source array and creates a new array with an additional dimension. It copies than the array ncopies times along the specified dimension.

A = [5, 3, 7, 4]

Then,

B = SPREAD (A, dim=1, ncopies=3)

results into

B = [5, 3, 7, 4]

$[5, 3, 7, 4]$
$[5, 3, 7, 4]$

Now the Algorithm for Gaussian elimination on the Connection machine using CM-Fortran is:

```
do i=1,n-1
  C = A(:,i) / A(i,i)
  C(1:i) = 0
  A = A - SPREAD (C, 2, n+1) * SPREAD (A(i,:),1,N)
end
```

7.7 Solving Partial Differential Equations

7.7.1 Finite Element Method

The finite element method solves a partial differential equation by replacing continuous functions by piecewise approximations. These approximations are defined as polygons referred as elements. Most common are polynomial approximations. The problem is than reduced to the problem to find solutions at the vertices of the polygons instead of the whole space. This can be done by either Gaussian elimination, conjugant gradient or multigrid methods.

7.7.2 Finite Difference Method

The finite difference method solves a partial differential equation on a discrete grid. The derivatives are approximated by the difference between the two grid points divided by the separation.

7.8 Laplace Equation

The Laplacian equation is described as follows:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

An example for the application for the Laplace equation is finding steady state voltage points in a 2 dimensional area.

7.8.1 Jacobi iteration

The Jacobi iteration scheme each trial solution is updated simultaneously as shown below.
Viewing a 2 dimensional area as a grid of points the Laplacian equation can be solved numerically. The value of each grid point $c_{i,j}$ can be calculated iteratively by the following formula:

$$c_{i,j}^{t-1} = c_{i,j-i}^t + c_{i,j+1}^t + c_{j-1,i}^t + c_{j+1,i}^t$$

The value is influenced by the value of its neighbor values on the grid. This calculation can naturally mapped in dataparallel fashion on a parallel SIMD machine.

7.8.2 Gauss Seidel iteration

In the Gauss Seidel iteration scheme the update is done sequentially in a particular order. Since the Jacobi iteration is slow in convergence it is of advantage to use the Gauss Seidel iteration. The Gauss Seidel scheme is a special case of the successive-over-relaxation (SOR) method. In this method a correction scheme is added to the basic term as used in the Jacobi iteration. the term consists of the old value multiplied by a weight factor.

$$c_{i,j}^{t-1} = c_{i,j-i}^{t} + c_{i,j+1}^{t} + c_{j-1,i}^{t} + c_{j+1,i}^{t} + (1-w)c_{i,j}^{t-1}$$

In case of Gauss Seidel w is 1. On an MIMD machine the domain is subdivided in equally big parts with overlapping regions at the borders (see Figures at the end of the section). To do the update correctly the Red-Black update scheme is used. This means that the domains mapped onto a processor are further subdivided into regions called black and red. First the sequential update is done on the red regions than on the black regions.
The overall algorithm looks like:

1. exchange edge values with neighbor processor
2. perform SOR update on red points in sequence
3. exchange edge values with neighbor processor
4. perform SOR update on black points in sequence
5. exchange edge values with neighbor processor
6. advance time step
7. if finish stop else goto 1

7.8.3 Data Parallel Program

The program has been successfully implemented on the CM2. The program outline is given in C.

```
#include <cscomm.h>
#define current Current
#define volatile
#include<cm/display.h>
#undef volatile
#undef current
#include<stdlib.h>
#include<stdio.h>
#define xsize 512          /* specify the maximal size of the grid */
#define ysize 512
shape [xsize][ysize] PICTURE;
char:PICTURE image;
int:PICTURE x, y, null;
float:PICTURE c, up, down, left, right;
WriteImage(int i)                /* Write the image in ASCII format */
{
 float max;
   printf (”Time:   max >?= c;
 fc = (c *256.0) / max;
 image = fc;
 CMSR-write-to-display (&image);
}
main(){
  char user-input[80];
  int time;
  /* initialize the Graphics facilities */
  CMSR-select-display-menu   (8, xsize, ysize);
  /*   CMSR-display-set-color-map (”greyscale”); */
  CMSR-display-set-color-map (”rainbow”);
  printf("please wait transfer picture"); printf(newline);
  with(PICTURE){
  x = pcoord(0);        /* assign to each processor a x,y coordinate*/
  y = pcoord(1);
  do{
      cnvg-test = 0;
      /* from-grid-dim corresponds to mpshift */
      /* the parameters given bellow explain the ussage */
      /* mpshift(array,shift,dim) = from-grid-dim(c,null,dim-1,shift) */
      /* processors at the boundaries do not participate in
         certain shift operations
      */
      c-old = c;
      where (y >= 0)
        up   = from-grid-dim (&c, null, 0, -1);
      where (y < ysize-1)
        down = from-grid-dim (&c, null, 0,  1);
      where (x < xsize-1)
        right= from-grid-dim (&c, null, 1, 1);
      where (x >= 0)
        left = from-grid-dim (&c, null, 1, -1);
      /* the convergence test should only be done one the interior
         points since the boundary values should be fixed */
      where (x >= 0) and (x < xsize-1){
           and  (y >= 0) and (y < ysize-1)
        c = (up + down + left + right) / 4.0    ;
        convg-test = amax2(conv-test, abs (c-old - c));
        /* the function amax2can be simulated by
                     convg-test >?= abs(c-old -c);
        */
      }
  } until (convg-test <= convg);
  WriteImage(time);
}
```

```
  printf("Press return to continue"); printf(newline);
  gets  (user-input);
}
```

7.8.4 Message Passing Program

The message passing program is slightly more complicated. Looking at the formulation of the problem the procedure mpshift defines a high level message passing send and receive routine. Usually these routines have to be used in a low level fashioned way. Here syncronization is assumed to be done with the help of the mpshift routine. As a result the asynchronous and synchronous communication between the processors is hidden for the user of the mpshift routine.

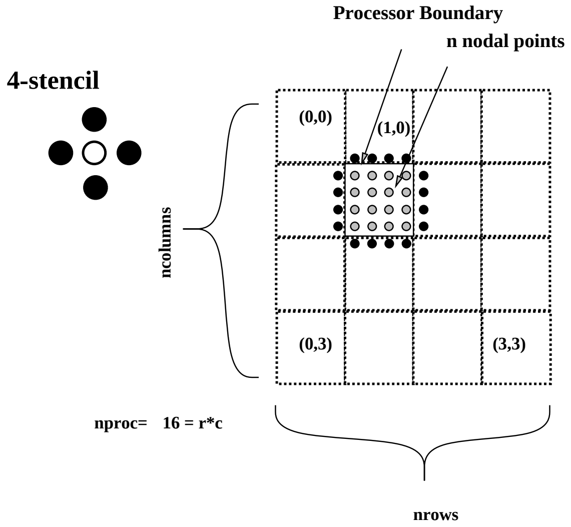


Figure 7.13: Decomposition. Example with 16 nodal points and 16 processors.

Figure 7.13 shows the decomposition of the program onto a grid of processing elements. Assume that each processor has a processor id of the form

$$(i, j) = pid$$

where i is the column in which the processor is located and j is the row. Assume the Processors are arranged in a processor grid with r rows and c columns. Than the number of processing elements is

$$nproc = c * r$$

The number of data elements in the problem is defined as

$$ntotal = nrows * ncolumns$$

Each processing element stores

$$nrows/r * ncolumns/c$$

elements, assuming nrows is a multiple of r and ncolumns is a multiple of c. To make the description of the algorithm easier, the following abbreviations are used:

```
#define mpshiftup    (source,dest,length) mpshift( 1,1,source,dest,length)
#define mpshiftdown (source,dest,length) mpshift(-1,1,source,dest,length)
#define mpshiftleft (source,dest,length) mpshift( 1,2,source,dest,length)
#define mpshiftright(source,dest,length) mpshift(-1,2,source,dest,length)
```

Now the algorithm can be defined as follows (note that the array index variables are starting from 0):

```
n-loc = nrows/r * ncolumns/c;
dimension f-new(nrows/r + 2, ncolumns/c + 2);
dimension f-old(nrows/r + 2, ncolumns/c + 2);
 indices starting from 0.
```

```
xdimension(f-old) = xdimension(f-new) = nrows/r+2
ydimension(f-old) = ydimension(f-new) = ncolumns/c+2
(row, column) = pid
main
   dest-up    = f-old(1,nrows/r+2)         — A
   dest-down  = f-old(1,0)                 — B
   dest-left  = f-old(ncolumns/c+2,1)      — C
   dest-right = f-old(0,1)                 — D
   source-up    = f-old(1,1)               — E
   source-down  = f-old(nrows/r,1)         — F
   source-left  = f-old(1,ncolumns/c)      — G
   source-right = f-old(1,1)               — H
   This results in the following relation ship between the processors:
     destination    Source
     x BBBBBBBB x   x ........ x   x ........ x
     D ........ C   . AAAAAAAA .   . C......D .
     D ........ C   . ........ .   . C......D .
     D ........ C   . BBBBBBBB .   . C......D .
     x AAAAAAAA x   x ........ x   x ........ x
    Initialize the addresses for the shift operation
   if (row = 0) then
     ymin = 1 ;
     dest-up = dummy
     source-down = dummy
   end if
   if (row = r) then
     ymax = ydimension(f-old) - 2;
     dest-down = dummy
     source-up = dummy
   end if
   if (column = 0) then
     xmin = 1;
     dest-left = dummy
     source-right = dummy
```

```
end if
if (column = r) then
 xmax = ximension(f-old) - 2;
 dest-right = dummy
 source-left = dummy
end if
begin
    test = 0
    shift to the right
      mpshiftup (source-right, dest-right, nrows/r)
    shift to the left
      mpshiftup (source-left, dest-left, nrows/r)
     shift up
      mpshiftup (source-up, dest-up, ncloumns/c)
     shift down
      mpshiftup (source-down, dest-down, ncloumns/c)
    do i=xmin,xmax
        do j=ymin,ymax
            temp = 1/4 * (  f-old(i,j+1) + f-old(i,j-1)
                          + f-old(i+1,j) + f-old(i-1,j));
            test = max (test, abs(temp - f-old(i,j))
            f-new(i,j) = temp
        end do
    end do
    do i=1,xdimension(f-new)-2
       do j=1,xdimension(f-new)-2
          f-old(i,j) = f-new(i,j)
       end do
    end do
    if (test > cnvg)
            goto begin
    else
            stop
    end if
```

```
    end begin
end main
```

7.8.5 Performance of the Parallel Program

Let t_{calc} denote the time used for one floating point operation and t_{comm} be the time used for the transfer of one datum from one processor to another one. A floating point operation is an addition, a subtraction or a multiplication.

One Dimensional Case

Two-stencil

Let n denote the number of nodal points stored in each processor. Than in each step of the Jacobi update the following equation is calculated:

$$\phi_i^{(k)} = \frac{1}{4}(\phi_{i-x}^{(k-1)} + \phi_{i+x}^{(k-1)}) \tag{7.1}$$

the equation uses 1 addition and 1 multiplication, resulting in 2 floating-point operations. The number of boundary elements is 2. Therefore, only 2 elements have to be updated to complete the calculation. The Figure 7.14 a) illustrates the relation graphically. Thus, the communication overhead is given by

$$f_c = \frac{2t_{comm}}{2nt_{calc}} = \frac{t_{comm}}{nt_{calc}} \tag{7.2}$$

Four-stencil

Let n denote the number of nodal points stored in each processor. Than in each step of the Jacobi update the following equation is calculated:

$$\phi_i^{(k)} = C(A(\phi_{i-x}^{(k-1)} + \phi_{i+x}^{(k-1)}) - \phi_{i-2x}^{(k-1)} - \phi_{i+2x}^{(k-1)}) \tag{7.3}$$

where A and C are constants. The equation uses 1 addition, 2 subtractions and 2 multiplications, resulting in 5 floating-point operations. The number of boundary elements is 4. Therefore, 4 elements have to be updated to complete the calculation. The Figure7.14 b) illustrates the relation graphically.
Thus, the communication overhead is given by

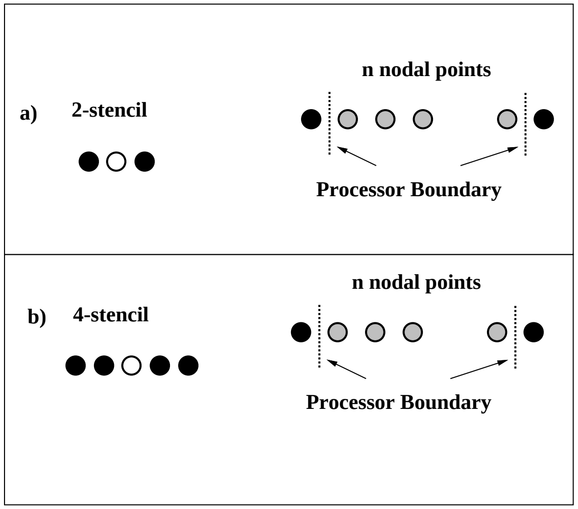


Figure 7.14: One dimensional case

$$f_c = \frac{4t_{comm}}{5nt_{calc}} = \frac{0.80t_{comm}}{nt_{calc}} \tag{7.4}$$

Two Dimensional Case

Four-stencil

Let n denote the number of nodal points stored in each processor. Than in each step of the Jacobi update the following equation is calculated:

$$\phi_i^{(k)} = \frac{1}{4}(\phi_{i-x}^{(k-1)} + \phi_{i+x}^{(k-1)} + \phi_{i-y}^{(k-1)} + \phi_{i+y}^{(k-1)}) \tag{7.5}$$

the equation uses 3 additions and 1 multiplication, resulting in 4 floating point operations. The number of boundary elements is $4\sqrt{n}$. Therefore, $4\sqrt{n}$ elements have to be updated to complete the calculation. The Figure 7.15 a) illustrates the relation graphically. Thus, the communication overhead is given by

$$f_c = \frac{4\sqrt{n}t_{comm}}{4nt_{calc}} = \frac{t_c omm}{\sqrt{n}t_{calc}} \tag{7.6}$$

Eight-stencil(distance 2)

Let n denote the number of nodal points stored in each processor. Than in each step of the Jacobi update the following equation is calculated:

$$\phi_i^{(k)} = \frac{1}{60}(16(\phi_{i-x}^{(k-1)} + \phi_{i+x}^{(k-1)} + \phi_{i-y}^{(k-1)} + \phi_{i+y}^{(k-1)}) - \phi_{i-2x}^{(k-1)} - \phi_{i+2x}^{(k-1)} - \phi_{i-2y}^{(k-1)} - \phi_{i+2y}^{(k-1)}) \tag{7.7}$$

The equation uses 3 additions, 4 subtractions and 2 multiplications, resulting in 9 floating point operations. The number of boundary elements is $8\sqrt{n}$. Therefore, $8\sqrt{n}$ elements have to be updated to complete the calculation. The Figure7.15 b) illustrates the relation graphically.
Thus, the communication overhead is given by

$$f_c = \frac{8\sqrt{n}t_{comm}}{9nt_{calc}} = \frac{0.89t_{comm}}{\sqrt{n}t_{calc}} \tag{7.8}$$

Eight-stencil (distance 1)

Let n denote the number of nodal points stored in each processor. Than in each step of the Jacobi update the following equation is calculated:

$$\phi_i^{(k)} = A(\phi_{i-x}^{(k-1)} + \phi_{i+x}^{(k-1)} + \phi_{i-y}^{(k-1)} + \phi_{i+y}^{(k-1)} + \phi_{i-x-y}^{(k-1)} + \phi_{i-x+y}^{(k-1)} + \phi_{i+x-y}^{(k-1)} + \phi_{i+x+y}^{(k-1)}) \tag{7.9}$$

The equation uses 7 additions and 1 multiplications, resulting in 8 floating point operations. The number of boundary elements is $4\sqrt{n}+4$. Therefore, $4(\sqrt{n}+1)$ elements have to be updated to complete the calculation. The Figure7.15 c) illustrates the relation graphically.
Thus, the communication overhead is given by

$$f_c = \frac{4(\sqrt{n}+1)t_{comm}}{8nt_{calc}} \tag{7.10}$$

For large n this leads to

$$f_c == \frac{0.5t_{comm}}{\sqrt{n}t_{calc}} \tag{7.11}$$

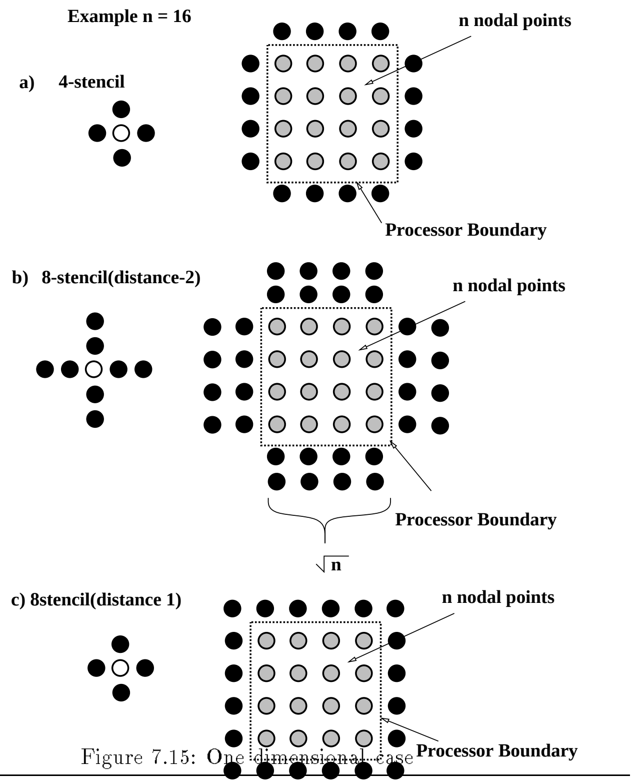


Figure 7.15: One dimensional case

3-dimensional Case

In a similar way the results for the three dimensional case can be obtained.

6-stencil

Let $n = a^3$ denote the number of nodal points stored in each processor. Than in each step of the Jacobi update the following equation is calculated:

$$\phi_i^{(k)} = A(\phi_{i-x}^{(k-1)} + \phi_{i+x}^{(k-1)} + \phi_{i-y}^{(k-1)} + \phi_{i+y}^{(k-1)} + \phi_{i-z}^{(k-1)} + \phi_{i+z}^{(k-1)}) \tag{7.12}$$

where A is a constant. The equation uses 5 additions and 1 multiplication, resulting in 6 floating point operations. The number of boundary elements is $6a^2$. Therefore, $6a^2$ elements have to be updated to complete the calculation. The Figure 7.16 a) illustrates the relation graphically. Thus, the communication overhead is given by

$$f_c = \frac{6a^2 t_{comm}}{6n t_{calc}} = \frac{t_{comm}}{a t_{calc}} \tag{7.13}$$

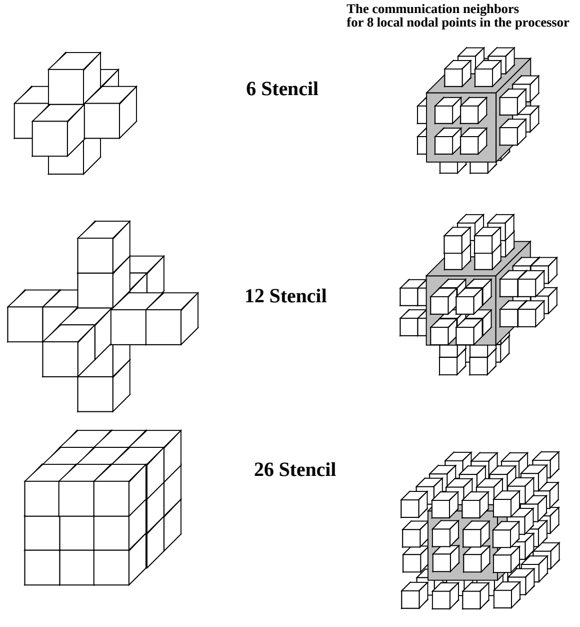

The communication neighbors
for 8 local nodal points in the processor
6 Stencil
12 Stencil
26 Stencil
one block is a nodal point

Figure 7.16: One dimensional case

Twelve-stencil

Let n=a*a*a denote the number of nodal points stored in each processor. Than in each step of the Jacobi update the following equation is calculated:

$$\phi_i^{(k)} = A(C(\phi_{i-x}^{(k-1)} + \phi_{i+x}^{(k-1)} + \phi_{i-y}^{(k-1)} + \phi_{i+y}^{(k-1)} + \phi_{i-z}^{(k-1)} + \phi_{i+z}^{(k-1)}) \tag{7.14}$$

$$- \phi_{i-2x}^{(k-1)} - \phi_{i+2x}^{(k-1)} - \phi_{i-2y}^{(k-1)} - \phi_{i+2y}^{(k-1)} \tag{7.15}$$

$$- \phi_{i-2z}^{(k-1)} - \phi_{i+2z}^{(k-1)}) \tag{7.16}$$

The equation uses 5 additions, 6 subtractions and 2 multiplications, resulting in 13 floating point operations. The number of boundary elements is twice as much as in the case before $12a^2$. Therefore, $12a^2$ elements have to be updated to complete the calculation. The Figure 7.16 b) illustrates the relation graphically. Thus, the communication overhead is given by

$$f_c = \frac{12a^2 t_{comm}}{13n t_{calc}} = \frac{0.923 t_{comm}}{a t_{calc}} \tag{7.17}$$

Twentysix-stencil

Let $n = a^3$ denote the number of nodal points stored in each processor. Than in each step of the Jacobi update the following equation is calculated:

$$\phi_i^{(k)} = A(C(\phi_{i-x}^{(k-1)} + \phi_{i+x}^{(k-1)} + \phi_{i-y}^{(k-1)} + \phi_{i+y}^{(k-1)} + \phi_{i-z}^{(k-1)} + \phi_{i+z}^{(k-1)})) \tag{7.18}$$

$$- \phi_{i-x-y-z}^{(k-1)} - \phi_{i-x-y+z}^{(k-1)} - \phi_{i-x+y-z}^{(k-1)} - \phi_{i-x+y+z}^{(k-1)} \tag{7.19}$$

$$- \phi_{i+x-y-z}^{(k-1)} - \phi_{i+x-y+z}^{(k-1)} - \phi_{i+x+y-z}^{(k-1)} - \phi_{i+x+y+z}^{(k-1)} \tag{7.20}$$

$$- \phi_{i-y-z}^{(k-1)} - \phi_{i-y+z}^{(k-1)} - \phi_{i+y-z}^{(k-1)} - \phi_{i+y+z}^{(k-1)} \tag{7.21}$$

$$- \phi_{i-x-z}^{(k-1)} - \phi_{i-x+z}^{(k-1)} - \phi_{i+x-z}^{(k-1)} - \phi_{i+x+z}^{(k-1)} \tag{7.22}$$

$$- \phi_{i-x-y}^{(k-1)} - \phi_{i-x+y}^{(k-1)} - \phi_{i+x-y}^{(k-1)} - \phi_{i+x+y}^{(k-1)} \tag{7.23}$$

The equation uses 5 additions, 20 subtractions and 2 multiplications, resulting in 27 floating point operations. The number of boundary elements is the number of boundary elements in the 6 Stencil plus the corner elements. Therefore, $6a^2 + 8 + 8a$ elements have to be updated to complete the calculation. The

Figure7.16 c) illustrates the relation graphically.

Thus, the communication overhead is given by

$$f_c = \frac{(6a^2 + 8a + 8)t_{comm}}{27n t_{calc}} = \frac{(6a^2 + 8a + 8)t_{comm}}{27a^3 t_{calc}} \tag{7.24}$$

For very large a the term is becomes

$$f_c = \frac{(6a^2)t_{comm}}{27n t_{calc}} = \frac{(0.22)t_{comm}}{a t_{calc}} \tag{7.25}$$

Bibliography

[1] Linpack Performance. database at netlib@ornl.gov.

[2] machines. database at netlib@ornl.gov.

[3] The Sirlin Report on Parallel Processing.

[4] CMMD User's Guide. Tech. rep., Thinking Machines Corporation, May 1993.

[5] SPEC Newsletter, 1993.

[6] Akl, S. G. *Parallel Sorting Algorithms.* Academic Press, 1985.

[7] Akl, S. G. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, New Jersy, 1989.

[8] Amdahl, G. M. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings* (1967), AFIPS Press.

[9] Anderson, E., Bai, Z., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. *Preliminary LAPACK Users' Guide.* Netlib, Oak Ridge National Laboratory, 1991.

[10] Anderson, E., Bai, Z., J.Dongarra, Greenbaum, A., McKenney, A., Croz, J. D., and Hammarling, S. LAPACK: A Portable Linear Algebra Library for High Performance Computers. *IEEE ...* (1990), 2–10.

[11] Anderson, E., Benzoni, A., Dongarra, J., Moulton, S., Ostrouchov, S., Tourancheau, B., , and van de Geijn, R. Basic Linear Algebra Communication Subprograms. In *Sixth Distributed Memory Computing Conference Proceedings, IEEE Computer Society Press* (1991), pp. pp. 287–290.

[12] Baiardi, F., and Orlando, S. Strategies for a massively parallel implementation of simulated annealing. In *Lecture Notes in Computer Science 366* (1989), Springer-Verlag, pp. pp. 273–287.

[13] Baily, D. Twelve Ways to Fool the Masses with Benchmarks. internet, monthley posting in comp.benchmark.

[14] Banerjee, P., Jones, M. H., and Sargent, J. Parallel simulated annealing for cell placement on hypercube multiprocessors. *? Vol. 1*, No. 1 (Jan 1990), pp. 91–106.

[15] Beguelin, A., Dongarra, J., Geist, G. A., Manchek, R., and Sunderam, V. A user's guide to PVM: Parallel virtual machine. Tech. Rep. TM-11826, Oak Ridge National Laboratory, 1991.

[16] Bell, C., and Newell, A. *Computer Structures: Readings and Examples.* McGraw-Hill, 1971.

[17] Bellocj, G. E., and Hardwick, J. C. Progamming Parallel Algorithms. Tech. Rep. CMU-CS-93-115, Carnegie Mellon University, Pittsburgh, PA, 1993.

[18] Bordawekar, R., del Rosario, J. M., and Choudhary, A. N. Design and Evaluation of Primitives for Parallel I/O. In *Supercomputing'93* (November 1993), IEEE Computer Society.

[19] Butler, R., and Lusk, E. User's guide to the p4 parallel programming system. Tech. rep., Argonne National Laboratory, Mathematics and Computer Science Division, October 1992.

[20] Cheng, D. Y. A Survey of Parallel Programming Languages and Tools. Tech. Rep. RND-93-005, NASA Ames Research Center, Moffet Field, CA, Mar. 1993.

[21] Dally, W. J. *A VLSI Architecture for Concurrent Data Strucures.* The Kluwer international series in engineering and computer science SECS 27. Kluwer Academic Publisher, Norwell, Massachusets, 1987.

[22] Davis, L. *Genetic Algorithms and Simulated Annealing.* Morgan Kaufmann, Los Altos, 1987.

[23] Dongara, J. Performance of various computers using standard linear equation software. Tech. Rep. CS-89-85, Oak Ridge National Laboratory, Mar. 1985.

[24] Dongarra, J., Croz, J. D., Hammarlin, S., and Duff, I. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software 16*, 1 (Mar 1990), pp. 1–17.

[25] Dongarra, J., Gustavson, F. G., and Karp, A. Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine. *SIAM Review 26*, 1 (Jan 1984), pp. 91–112.

[26] Dowd, K. High Performance Computing, June 1993.

[27] Faigle, U., and Schrade, R. Simulated Annealing – Eine Fallstudie. *Angewandte Informatik*, 6 (June 1988), pp. 259–263. (in german).

[28] Feng, T.-Y. Some Characteristics of Associative/Parallel Processing. pp. 5–16.

[29] Flynn, M. Very High-Speed Computing Systems. *Proceedings of the IEEE 54*, 12 (December 1966), 1901–1909.

[30] Flynn, M. J. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Computers C-21*, 9 (September 1972), 948–960.

[31] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. *Solving Problems on Concurrent Processors.* Prentice Hall, New Jersey, 1988.

[32] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. *Solving Problems on Concurrent Processors.* Prentice Hall, New Jersey, 1988.

[33] Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H. PICL: A portable instrumented communications library. Tech. Rep. TM-11130, Oak Ridge National Laboratory, 1990.

[34] Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H. PICL, A Portable Intrumented Communication Library, C Reference Manual. Tech. Rep. Tech. Rep. ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, July 1990.

[35] Gilbert, J. R., and Zmijewski, E. A Parallel Graph Partitioning Algorithm for a Message-Passing Multiprocessor. In *1st Int. Conf. on Supercomputing* (1987), pp. 498–512.

[36] Gilbert, J. R., and Zmijewski, E. A Parallel Graph Partitioning Algorithm for a Message-Passing Multiprocessor. Tech. Rep. 87-803, Cornell University, 1987.

[37] Giloi, W. A Complete Taxonomy of Computer Architecture Based on the Abstract Data Type View. In *IFIP Workshop on Taxonomy in Computer Architecture* (June 1981), pp. 19–38.

[38] Golub, G. H., and Loan, C. F. V. *Matrix Computations.* John Hopkins University Press, 1989.

[39] Greening, D. R. Parallel simulated annealing techniques. *Physica D 42* (1990), pp. 293–306.

[40] Händler, W. The Impact of Classification Schemes on Computer Architecture. In *1977 International Conference on Parallel Processing* (August 1977), pp. 7–15.

[41] Händler, W. Standards, Classification, and Taxonomy: Experiences with ECS. In *IFIP Workshop on Taxonomy in Comp;uter Architecture* (June 1981), pp. 39–75.

[42] Händler, W. Innovative Computer Architecture - How to Increase Parallelism but not Complexity. In *Parallel Processing Systems - An Advanced Course*, D. J. Evans, Ed. Cambridge University Press, 1982, pp. 1–42.

[43] Herrarte, V., and Lusk, E. Studying parallel program behavior with Upshot. Tech. Rep. TM-11130, Oak Ridge National Laboratory, 1991.

[44] Hillis, W. D., and Steele, G. L. Data Parallel Programming. *Communications of the ACM 29*, 12 (December 1986), 1170–1183.

[45] Hoare, C. A Model for Communicating Sequential Processes. Tech. Rep. PRG-22, Oxford University Programming Research Group, 1981.

[46] Hockney, R., and Berry, M. Public International Benchmarks for Parallel Computers. Tech. rep., PARKBENCH Committee, University Tennessee, Nov. 1993.

[47] Hockney, R., and Jesshope, C. *Parallel Computers.* Adam Hilger, 1981.

[48] Hwang, K., and Briggs, F. A. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1985.

[49] Hwang, K., and Xu, J. Mapping Partitioned Program Modules onto Multicomputer Nodes Using Simulated Annealing. In *International Conference Parallel Processing* (1990), vol. Vol. 2, pp. pp. 292–293.

[50] Johnson, D. S., Aragon, C. R., and McGeoch, L. A. Optimization by Stimulated Annealing: An Experimental EvaluationPart I, Graph Partitioning. *Operations research Vol. 37*, No. 6 (Nov. 1989).

[51] Kernighan, B., and Lin, S. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal Vol. 49*, No. 2 (1970), pp. 291–308.

[52] Kernighan, B. W., and Lin, S. An Efficient Heuristic Algorithm for the Traveling Salesman Problem. *Operations Research 21* (1973), 498–516.

[53] Kirkpatrick, S. Optimization by Simulated Annealing: Quantitative Studies. *Journal of Statistical Physics 34* (1984), pp. 975–986.

[54] Kirkpatrick, S., Gellatt, C. D., and Vecchi, M. P. Optimization by Simulated Annealing. *Science 220* (1983), pp. 671–680.

[55] Knuth, D. E. *The Art of Computer Programming, Sorting and Searching*, vol. Volume 3. Addison Wesley, 1973.

[56] Landau, R. H., and Fink, P. J. *A scientist's and enggineer's guide to Workstations and Supercomputers*. Wiley-Interscience, 1993.

[57] M. Heath, J. A. E. Paragraph. Tech. rep., Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, Oct 1991.

[58] MasPar Computer Corporation. *MasPar System Overview*, pn 9300-9001-00, revision a5 ed., Aug. 1991.

[59] Miyata, E. Frequent asked Questions: Benchmark. posted monthly in newsgroup comp.benchmark.

[60] Mohamed, A. G. Block-based Solvers for Engineering Applications. In *Mechanics Computing in the 1990's and Beyond, Proceedings of the ASCE Engineering Mechanics Speciality Conference* (Columbus, Ohio, May 1991), H. Adeli and R. L. Sierakowski, Eds., ASCE, New York, pp. pp. 19–22.

[61] Mohamed, A. G., Fox, G. C., and von Laszewski, G. Blocked LU Factorization on a Multiprocessor Computer. *Microcomputer in Civil Engeneering Vol. 8*, No. 1 (1993), pp. 45–56.

[62] Moore, D. A Round-Robin Parallel Partitioning Algorithm. Tech. Rep. 88–916, Cornell University, 1988.

[63] Moreau, R. *The Computer Comes of Age*. MIT Press, 1984.

[64] Ortega, J. M. *Introduction to Parallel and Vector Solution of Linear Systems*. Frontiers of Computer Science. Plenum Press, New York, 1988.

[65] Papadimitriou, C. H., and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall Inc., 82.

[66] Raman, S., and Patnik, L. Circut Partitioning using Simulated Annealing on a Hypercube. Tech. rep., Coordinated Science Laboppratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA, 1988.

[67] Rice, J. R. *Numerical Methods, Software, and Analysis*. McGraw-Hill, 1983.

[68] Rosen, S. Electronic Computers: A Historical Survey. *Computing Surveys 1*, 7 (1960).

[69] Smith, B. Parallel Programming Tools. Tech. Rep. Draft, Argonne National Laboratory, University of California, Los Angeles, February 1993.

[70] Sterlin. Sterlin Report on Parallel Processing, 1993.

[71] Trew, A., and eds., G. W. *Past, Present, Future*. Springer, 1991.

[72] van de Geijn, R. Massively Parallel LINPACK Benchmark on the Intel Touchstone DELTA and iPSC/860 Systems: Progress Report. Tech. Rep. Computer Science Technical Report TR-91-28, University of Texas, Aug, updated Dec 5 1991.

[73] von Laszewski, G. A parallel genetic algorithm for the graph partitioning problem. In *Transputer Research and Applications 4, Proc. of the 4th Conf. of the North-American Transputers Users Group* (1990), IOS Press.

[74] von Laszewski, G. Intelligent Structural Operators for the k-way Graph Partitioning Problem, July 1991. plenary presentation.

[75] von Laszewski, G. A Collection of Graph Partitioning Algorithms: Simulated Annealing, Simulated Tempering, Kernighan Lin, Two Optimal, Graph Reduction, Bisection. Tech. Rep. SCCS 477, Northeast Parallel Architectures Center at Syracuse University, April 1993.

[76] von Laszewski, G. Object Oriented Recursive Bisection on the CM-5. Tech. Rep. SCCS 476, Northeast Parallel Architectures Center at Syracuse University, April 1993.

[77] von Laszewski, G., and Mühlenbein, H. A Parallel Genetic Algorithm for the k-way Graph Partitioning Problem. In *1st inter. Workshop on Parallel Problem Solving from Nature* (Nov. 1990), Springer, Ed.

[78] von Laszewski, G., Parashar, M., Mohamed, A. G., and Fox, G. C. High Performance Scalable Matrix Algebra Algorithms for Distributed Memory Architectures. Tech. Rep. SCCS 271b, Northeast Parallel Architectures Center at Syracuse University, CRPC-TR92210, Center for Research on Parallel Computation, Rice University, Houston, TX, June 1992.

[79] von Laszewski, G., Parashar, M., Mohamed, A. G., and Fox, G. C. High Performance Scalable Matrix Algebra Algorithms for Distributed Memory Architectures. In *Proceedings of Supercomputing 92* (Minneapolis, Nov. 1992), IEEE Compt. Soc. Press, pp. 170–179. Best Student Paper Award.

[80] Xu, J., and Hwang, K. Simulated annealing method for mapping production systems onto multicomputers. In *IEEE Conf AI Applic* (1990), pp. pp. 350–356.