

An Introduction to Parallel Programming

An Introduction to Parallel Programming

Tobias Wittwer

VSSD

© **Tobias Wittwer**

First edition 2006

Published by:

VSSD

Leeghwaterstraat 42, 2628 CA Delft, The Netherlands

tel. +31 15 278 2124, telefax +31 15 278 7585, e-mail: hlf@vssd.nl

internet: <http://www.vssd.nl/hlf>

URL about this book: **<http://www.vssd.nl/hlf/a019.htm>**

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

ISBN-10 90-71301-78-8

ISBN-13 978-90-71301-78-0

NUR 958

Keywords: parallel programming

Foreword

Any computer user knows the craving for more computing power. It was this “need for speed” that made me choose high performance computing as subject of my master thesis. This gave me the opportunity to work with systems at Stuttgart’s high performance computing center, HLRS. I spent five months parallelising programs for gravity field modelling using OpenMP and MPI, and testing them on a Cray Opteron cluster, NEC TX-7, and NEC SX-6.

After graduating I moved to the Netherlands for my PhD research in the Physical and Space Geodesy Group of the Delft Institute of Earth Observation and Space System, at the faculty of Aerospace Engineering of the Delft University of Technology. My research topic, regional gravity field modelling, proved to be computationally intensive. Luckily, we had access to Teras and Aster, two supercomputers at the SARA supercomputing facility in Amsterdam. My programs were quickly parallelised, shortening program runs from hours to minutes.

Seeing my colleagues struggle with the limited power of their PCs gave me the idea of writing a tutorial about parallel programming, to give everyone the opportunity to easily parallelise her or his programs. More urge was added when our group decided to buy its own Linux cluster. Now all I needed was an example program - and when I had to write a program for spherical harmonic analysis to check some results, I had this as well.

A week of coding and writing ensued. Test computations, creating graphics, proofreading, and finetuning followed. My supervisor Prof. Dr.-Ing. Roland Klees reviewed the document and gave his approval. I hope that the finished product will be useful to the reader, and as enjoyable to read as it was to write.

Tobias Wittwer, Delft, November 2006

Contents

1	Introduction	1
1.1	Goal	1
1.2	Prerequisites	2
1.3	Example Program	2
2	System Architectures	3
2.1	Single Instruction - Single Data (SISD)	3
2.2	Single Instruction - Multiple Data (SIMD)	5
2.3	Multiple Instruction - Multiple Data (MIMD)	5
2.4	Shared Memory	6
2.5	Distributed Memory	7
2.6	ccNUMA	8
2.7	Cluster	9
2.8	Multiple Instruction - Single Data (MISD)	10
2.9	Some Examples	11
3	Software	15
3.1	Compiler	15
3.2	BLAS & LAPACK	17
3.3	MPI	19
3.4	BLACS	21
3.5	ScaLAPACK	21

4	Performance Analysis	23
4.1	Timing	23
4.2	Profiling	24
4.3	Measuring Performance	24
5	SHALE - a program for spherical harmonic analysis	27
5.1	Spherical harmonic analysis	27
5.2	Direct method	29
5.3	Conjugate gradient method	40
5.4	Conclusions	48
	Bibliography	51
	Index	53

Chapter 1

Introduction

1.1 Goal

Many scientific computations require a considerable amount of computing time. This computing time can be reduced by distributing a problem over several processors. Multiprocessor computers used to be quite expensive, and not everybody had access to them. Since 2005, x86-compatible CPUs designed for desktop computers are available with two “cores”, which essentially makes them dualprocessor systems. More cores per CPU are to follow.

This cheap extra computing power has to be used efficiently, which requires parallel programming. Parallel programming methods that work on dual-core PCs also work on larger shared memory systems, and a program designed for a cluster or other type of distributed memory system will also perform well on your dual-core (or multi-core) PC.

The goal of this tutorial is to give an introduction into all aspects of parallel programming that are necessary to write your own parallel programs. To achieve this, it explains

- the various existing architectures of parallel computers,
- the software needed for parallel programming, and how to install and configure it,
- how to analyse software and find the points where parallelisation might be helpful,
- how to write parallel programs for shared memory computers using OpenMP,
- how to write parallel programs for distributed memory computers using MPI and ScaLAPACK.

This tutorial aims mainly at writing parallel programs for solving linear equation systems. I hope that it is also useful to give some help for parallelising programs for other applications.

1.2 Prerequisites

This introduction to parallel programming assumes that you

- work under Linux, as it is the most common platform for high performance computing,
- use the Intel `ifort` or GNU `gfortran` Fortran compiler, as these are freely available (Intel only for non-commercial purposes) OpenMP-capable compilers.

This tutorial should also be useful for people using different system configurations and/or programming languages. All the examples use above mentioned configuration, but can easily be adapted to other configurations. The example programs are written in Fortran, but should also be understandable for C programmers. OpenMP and MPI work very similar in C/C++, with only a slightly different syntax (`#OMP PARALLEL FOR` instead of `!$OMP PARALLEL DO`, different argument types). For ScaLAPACK programming, I recommend using Fortran, as ScaLAPACK can be a little awkward to use with C/C++.

1.3 Example Program

The example program SHALE implements spherical harmonical analysis (SHA) using least-squares estimation of the spherical harmonic coefficients. I consider SHA to be well suited as an example, as it is quite simple and understandable, can be run in various problem sizes, and offers several starting points for parallel implementation. The functional model can easily be exchanged for other functional models, making SHALE a good example for your own parallelised parameter estimation programs.

All versions of SHALE described in this tutorial are available from the author's website, <http://www.lr.tudelft.nl/psg> → Staff → Tobias Wittwer → personal homepage.

Chapter 2

System Architectures

A system for the categorisation of the system architectures of computers was introduced by Flynn (1972). It is still valid today and cited in every book about parallel computing. It will also be presented here, expanded by a description of the architectures actually in use today.

2.1 Single Instruction - Single Data (SISD)

The most simple type of computer performs one instruction (such as reading from memory, addition of two values) per cycle, with only one set of data or operand (in case of the examples a memory address or a pair of numbers). Such a system is called a *scalar computer*.



Figure 2.1: Summation of two numbers

Figure 2.1 shows, in a simplified manner, the summation of two numbers in a scalar computer. As a scalar computer performs only one instruction per cycle, five cycles are needed to complete the task - only one of them dedicated to the actual addition. To add n pairs of numbers, $n \cdot 5$ cycles would be required. To make matters even worse, in reality each of the steps shown in figure 2.1 is actually composed of several sub-steps, increasing the number of cycles required for one summation even more.

The solution to this inefficient use of processing power is *pipelining*. If there is one functional unit available for each of the five steps required, the addition still requires five cycles. The

advantage is that with all functional units being busy at the same time, one result is produced every cycle. For the summation of n pairs of numbers, only $(n - 1) + 5$ cycles are then required. Figure 2.2 shows the summation in a pipeline.

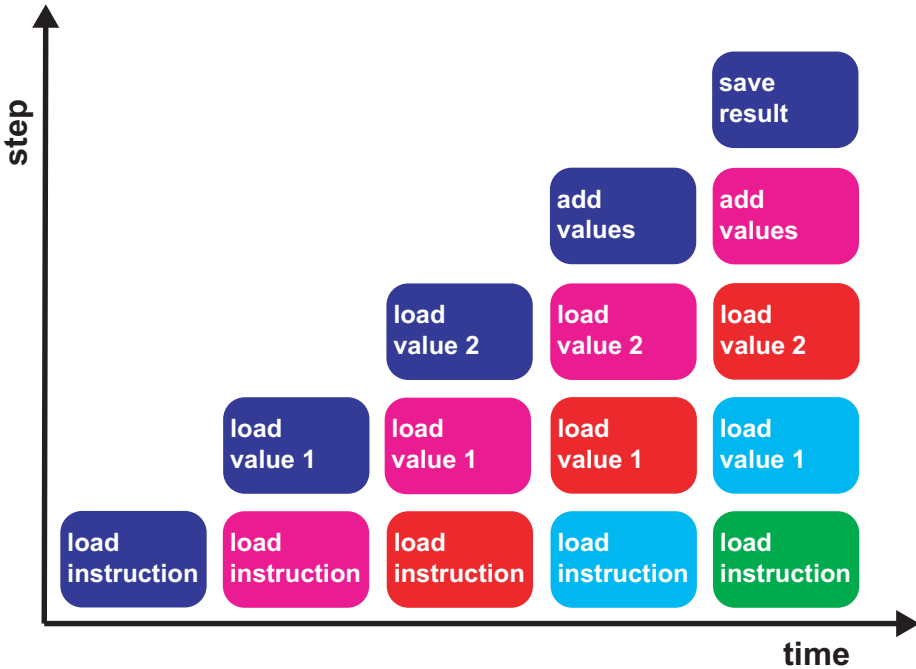


Figure 2.2: Summation of two numbers in a pipeline

As the execution of instructions usually takes more than five steps, pipelines are made longer in real processors. Long pipelines are also a prerequisite for achieving high CPU clock speeds. These long pipelines generate a new problem. If there is a branching event (such as due to an *if*-statements), the pipeline has to be emptied and filled again, and there is a number of cycles equal to the pipeline length until results are again delivered. To circumvent this, the number of branches should be kept small (avoiding and/or smart placement of *if*-statements). Compilers and CPUs also try to minimise this problem by “guessing” the outcome (branch prediction).

The power of a processor can be increased by combining several pipelines. This is then called a *superscalar* processor. Fixed-point and logical calculations (performed in the *ALU* - Arithmetic/Logical Unit) are usually separated from floating-point math (done by the *FPU* - Floating Point Unit). The FPU is commonly subdivided in a unit for addition and one for multiplication. These units may be present several times, and some processors have additional functional units for division and the computation of square roots.

To actually gain a benefit from having several pipelines, these have to be used at the same time. Parallelisation is necessary to achieve this.

2.2 Single Instruction - Multiple Data (SIMD)

The scalar computer of the previous section performs one instruction on one data set only. With numerical computations, we often handle larger data sets on which the same operation (the same instruction) has to be performed. A computer that performs one instruction on several data sets is called a *vector* computer.

Vector computers work just like the pipelined scalar computer of figure 2.2. The difference is that instead of processing single values, vectors of data are processed in one cycle. The number of values in a vector is limited by the CPU design. A vector processor that can simultaneously work with 64 vector elements can also generate 64 results per cycle - quite an improvement over the scalar processor from the previous section, which would require at least 64 cycles for this.

To actually use the theoretically possible performance of a vector computer, the calculations themselves need to be vectorised. If a vector processor is fed with single values only, it cannot perform decently. Just like with a scalar computer, the pipelines need to be kept filled.

Vector computers used to be very common in the field of high performance computing, as they allowed very high performance even at lower CPU clock speeds. In the last years, they have begun to slowly disappear. Vector processors are very complex and thus expensive, and perform poorly with non-vectorisable problems. Today's scalar processors are much cheaper and achieve higher CPU clock speeds. Vectorisation is not dead, though. With the Pentium III, Intel introduced *SSE* (Streaming SIMD Extensions), which is a set of vector instructions. In certain applications, such as video encoding, the use of these vector instructions can offer quite impressive performance increases. More vector instructions were added with SSE2 (Pentium 4) and SSE3 (Pentium 4 Prescott).

2.3 Multiple Instruction - Multiple Data (MIMD)

Up to this point, we only considered systems that process just one instruction per cycle. This applies to all computers containing only one processing core (with multi-core CPUs, single-CPU systems can have more than one processing core, making them MIMD systems). Combining several processing cores or processors (no matter if scalar or vector processors) yields a computer that can process several instructions and data sets per cycle. All high performance computers belong to this category, and with the advent of multi-core CPUs, soon all computers will. MIMD systems can be further subdivided, mostly based on their memory architecture.

2.4 Shared Memory

In MIMD systems with shared memory (SM-MIMD), all processors are connected to a common memory (RAM - Random Access Memory), as shown in figure 2.3. Usually all processors are identical and have equal memory access. This is called *symmetric multiprocessing* (SMP).

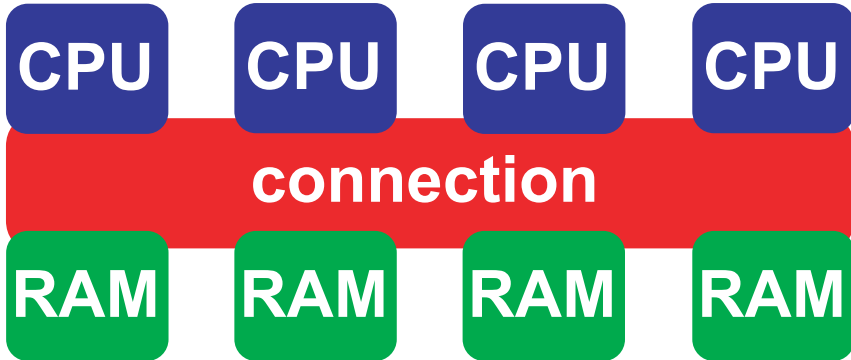


Figure 2.3: Structure of a shared memory system

The connection between processors and memory is of predominant importance. Figure 2.4 shows a shared memory system with a bus connection. The advantage of a bus is its expandability. A huge disadvantage is that all processors have to share the bandwidth provided by the bus, even when accessing different memory modules. Bus systems can be found in desktop systems and small servers (frontside bus).

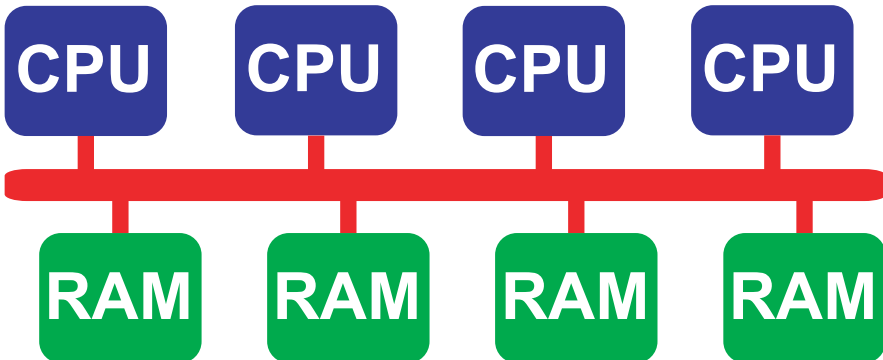


Figure 2.4: Shared memory system with bus

To circumvent the problem of limited memory bandwidth, direct connections from each CPU to

each memory module are desired. This can be achieved by using a crossbar switch (figure 2.5). Crossbar switches can be found in high performance computers and some workstations.

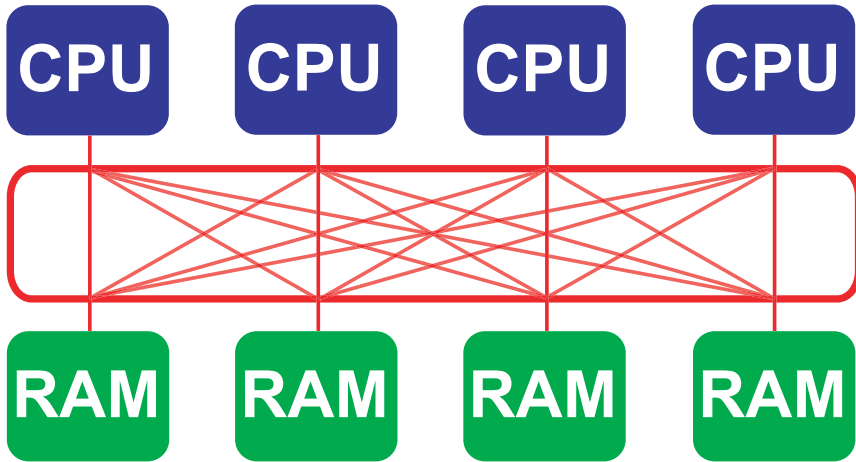


Figure 2.5: Shared memory system with crossbar switch

The problem with crossbar switches is their high complexity when many connections need to be made. This problem can be weakened by using multi-stage crossbar switches, which in turn leads to longer communication times. For this reason, the number of CPUs and memory modules than can be connected by crossbar switches is limited.

The big advantage of shared memory systems is that all processors can make use of the whole memory. This makes them easy to program and efficient to use. The limiting factor to their performance is the number of processors and memory modules that can be connected to each other. Due to this, shared memory-systems usually consist of rather few processors.

2.5 Distributed Memory

As could be seen in the previous section, the number of processors and memory modules cannot be increased arbitrarily in the case of a shared memory system. Another way to build a MIMD-system is distributed memory (DM-MIMD).

Each processor has its own local memory. The processors are connected to each other (figure 2.6). The demands imposed on the communication network are lower than in the case of a shared memory system, as the communication between processors may be slower than the communication between processor and memory.

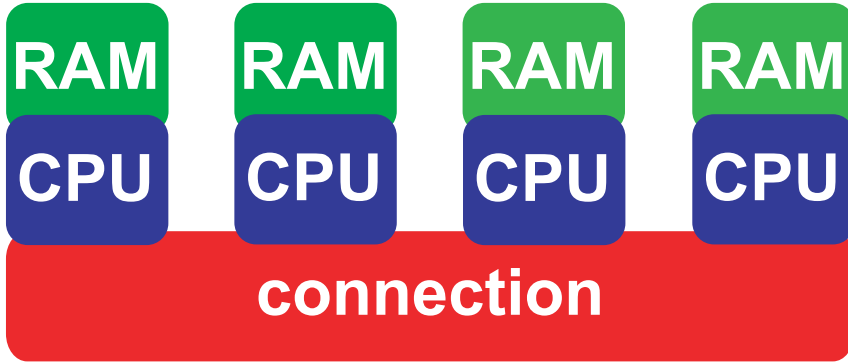


Figure 2.6: Structure of a distributed memory system

Distributed memory systems can be hugely expanded. Several thousand processors are not uncommon, this is called *massively parallel processing* (MPP). To actually use the theoretical performance, much more programming effort than with shared memory systems is required. The problem has to be subdivided into parts that require little communication. The processors can only access their own memory. Should they require data from the memory of another processor, then these data have to be copied. Due to the relatively slow communications network between the processors, this should be avoided as much as possible.

2.6 ccNUMA

The two previous sections showed that shared memory systems suffer from a limited system size, while distributed memory systems suffer from the arduous communication between the memories of the processors. A compromise is the ccNUMA (cache coherent non-uniform memory access) architecture.

A ccNUMA system (figure 2.7) basically consists of several SMP systems. These are connected to each other by means of a fast communications network, often crossbar switches. Access to the whole, distributed or non-unified memory is possible via a common cache.

A ccNUMA system is as easy to use as a true shared memory system, at the same time it is much easier to expand. To achieve optimal performance, it has to be made sure that local memory is used, and not the memory of the other modules, which is only accessible via the slow communications network. The modular structure is another big advantage of this architecture. Most ccNUMA system consist of modules that can be plugged together to get systems of various sizes.

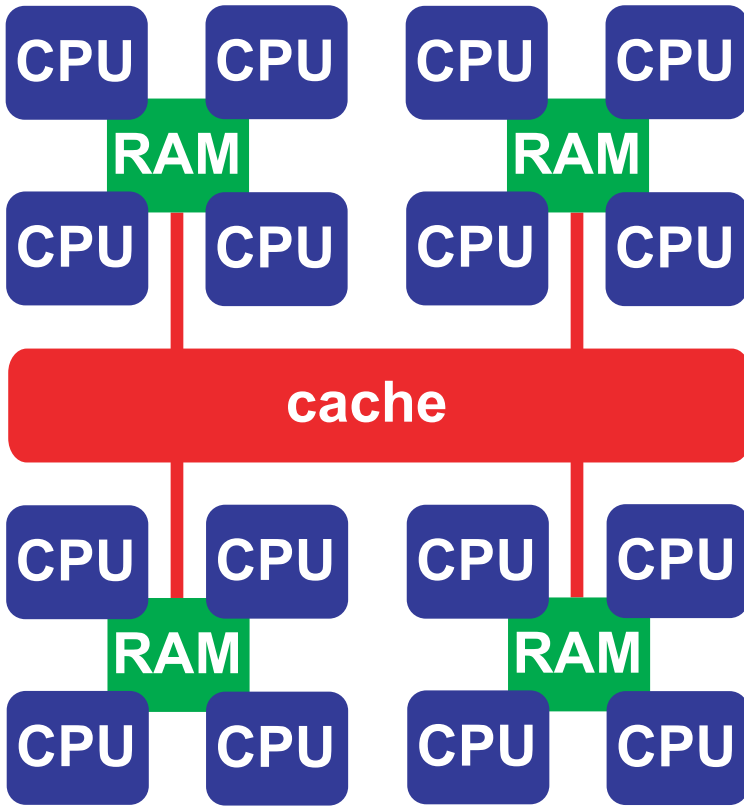


Figure 2.7: Structure of a ccNUMA system

2.7 Cluster

For some years now clusters are very popular in the high performance computing community. A cluster consists of several cheap computers (nodes) linked together. The simplest case is the combination of several desktop computers - known as a *network of workstations* (NOW). Most of the time, SMP systems (usually dual-CPU system with Intel or AMD CPUs) are used because of their good value for money. They form *hybrid* systems. The nodes, which are themselves shared memory systems, form a distributed memory system (figure 2.8).

The nodes are connected via a fast network, usually *Myrinet* or *Infiniband*. Gigabit Ethernet has approximately the same bandwidth of about 100 MB/s and is a lot cheaper, but the latency (travel time of a data package) is much higher. It is about 100 μ s for Gigabit Ethernet compared to only 10 - 20 μ s for Myrinet. Even this is a lot of time. At a clock speed of 2 GHz, one cycle takes 0.5

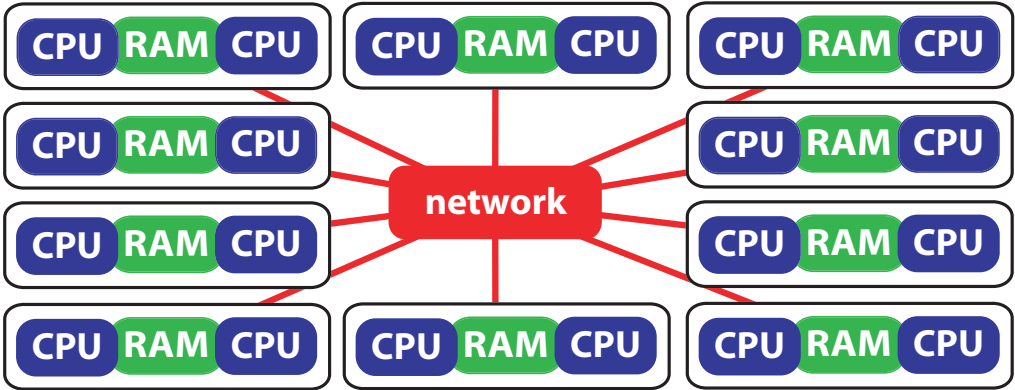


Figure 2.8: Structure of a cluster of SMP nodes

ns. A latency of $10\ \mu\text{s}$ amounts to 20,000 cycles of travel time before the data package reaches its target.

Clusters offer lots of computing power for little money. It is not that easy to actually use the power. Communication between the nodes is slow, and as with conventional distributed memory systems, each node can only access its local memory directly. The mostly employed PC architecture also limits the amount of memory per node. 32 bit systems cannot address more than 4 GB of RAM, and x86-64 systems are limited by the number of memory slots, the size of the available memory modules, and the chip sets. Despite these disadvantages, clusters are very successful and have given traditional, more expensive distributed memory systems a hard time. They are ideally suited to problems with a high degree of parallelism, and their modularity makes it easy to upgrade them.

In recent years, the cluster idea has been expanded to connecting computers all over the world via the internet. This makes it possible to aggregate enormous computing power. Such a widely distributed system is known as a *grid*.

2.8 Multiple Instruction - Single Data (MISD)

The attentive reader may have noticed that one system architecture is missing: *Multiple Instruction - Single Data* (MISD). Such a computer is neither theoretically nor practically possible in a sensible way. Openshaw et al. (1999) write: “We found it hard to figure out why you would want to do this (the simultaneous manipulation of one data set with several operations) unless you are a computer scientist interested in weird computing! It is a highly specialised and seemingly a

very restrictive form of parallelism that is often impractical, not to mention useless, as the basis for a general-purpose machine.”

2.9 Some Examples

This section presents a few common multiprocessor/multi-core architectures. A much more extensive and detailed description is given in the “Overview of recent supercomputers”, which is updated once a year, and available on-line at <http://www.phys.uu.nl/~euroben/>.

Twice a year, a list of the 500 fastest computers in the world is published. The ranking is based on the LINPACK benchmark. Although this is an old benchmark with little practical reference, the Top 500 list gives a good overview of the fastest computers and the development of supercomputers. The list can be viewed on-line at <http://www.top500.org>.

2.9.1 Intel Pentium D

The Intel Pentium D was introduced in 2005. It is Intel’s first dual-core processor. It integrates two cores, based on the NetBurst design of the Pentium 4, on one chip. The cores have their own caches and access the common memory via the frontside bus. This limits memory bandwidth and slows the system down in the case of memory-intensive computations. The Pentium D’s long pipelines allow for high clock frequencies (at the time of writing up to 3.73 GHz with the Pentium D 965), but may cause poor performance in the case of branches. The Pentium D is not dual-CPU-capable. This capability is reserved for the rather expensive Xeon CPU. The Pentium D supports SSE3 and x86-64 (the 64bit-extension of the x86 instruction set).

2.9.2 Intel Core 2 Duo

Intel’s successor to the Pentium D is similar in design to the popular Pentium M design, which in turn is based on the Pentium III, with ancestry reaching back to the Pentium Pro. It abandons high clock frequencies in the favour of more efficient computation. Like the Pentium D, it uses the frontside bus for memory access by both CPUs. The Core 2 Duo supports SSE3 and x86-64.

2.9.3 AMD Athlon 64 X2 & Opteron

AMD’s dual-core CPUs Athlon 64 X2 (single-CPU only) and Opteron (depending on model up to 8 CPUs in one system possible) are very popular CPUs for Linux clusters. They offer good performance at affordable prices and reasonable power consumption. Each core has its

own HyperTransport channel for memory access, making these CPUs well suited for memory-intensive applications. They also support SSE3 and x86-64.

2.9.4 IBM pSeries

The pSeries is IBM's server- and workstation line based on the POWER processor. The newer POWER processors are multi-core designs and feature large caches. IBM builds shared memory systems with up to 32 CPUs. One large pSeries installation is the JUMP cluster at Kernforschungszentrum Jülich, Germany (<http://jumpdoc.fz-juelich.de>).

2.9.5 IBM BlueGene

BlueGene is an MPP (massively parallel processing) architecture by IBM. It uses rather slow 700 MHz PowerPC processors. These processors form very large, highly integrated distributed memory systems, with fast communication networks (a 3D-Torus, like the Cray T3E). At the time of writing, position one and three of the Top 500 list were occupied by BlueGene systems. The fastest system, BlueGene/L (http://www.llnl.gov/asc/computing_resources/bluegene1/), consists of 131,072 CPUs, and delivers a performance of up to 360 TeraFLOPS.

2.9.6 NEC SX-8

The NEC SX-8 is the one of the few vector supercomputers in production at the moment. It performs vector operations at a speed of 2 GHz, with eight operations per clock cycle. One SX-8 node consists of eight CPUs, up to 512 nodes can be connected. The biggest SX-8 installation is, at the time of writing, the 72-node system at Höchstleistungsrechenzentrum Stuttgart (HLRS), Germany.

2.9.7 Cray XT3

Cray is the most famous name in supercomputing. Many of its designs were known not only for their performance, but also for their design. The Cray XT3 is a massively-parallel system using AMD's Opteron CPU. The biggest installation of an XT3 is "Red Storm" at Sandia National Laboratories (<http://www.sandia.gov/ASC/redstorm.html>) with 26,544 dual-core Opteron CPUs, good for a performance of more than 100 TFLOPS and the second position in the November 2006 Top 500 list.

2.9.8 SGI Altix 3700

The SGI Altix 3700 is a ccNUMA system using Intel's Itanium 2 processor. The Itanium 2 has large caches and good floating point performance. Being ccNUMA, the Altix 3700 is easy to program. Aster at SARA, Amsterdam, the Netherlands (<http://www.sara.nl/userinfo/aster/description/index.html>) is an Altix 3700 with 416 CPUs.