Overview of State-of-the-art High-performance Computer Systems

Fridolin Hofmann
Chair of Computer Science (Operating Systems)
University of Erlangen-Nuremberg, Germany

16 Mar 1995

Abstract

The power of high performance computing systems is largely obtained by executing several computations in parallel. Full use of these systems can only be made if application programs are implicitly or explicitly characterized by a sufficient amount of parallelism of the kind the computing system offers. When programming high performance systems one has not only to consider the structure of the hardware but also the structure of their operating systems and the so called programming models given by programming languages and communications libraries. Therefore, at first we give an overview over the essential features of state of the art hardware followed by a discussion of their operating systems. Finally we address the influence of the different programming models.

[Slide 1]

1 Introduction

[Slide 2]

Frequently the reasons given for high-performance computing are the so called Grand Challenges. In a more technical sense it may be defined as solving problems that need today when using traditional algorithms and monoprocessors

- too much computing power
- too much primary storage
- too high transfer rates between primary and secondary storage.

There are two ways to overcome to a high degree the present limits.

The first one is to develop better algorithms. Convincing successes in solving partial differential equations for example are the multigrid methods, the sparse grid methods and recently the active set method. Compared to former methods the number of required operations could be reduced to 1% or even to 0.1%.

The other possibility is to divide the task or parts of it into several independent subtasks and solving them in parallel. This may be done for example by using several interconnected processors.

[Slide 3]

Looking for possibilities to do computations or parts of them in parallel one finds two classes of mechanisms, namely pipelining and concurrency.

Pipelining tries to subdivide operations into a seqeunce of functionally independent suboperations. If the operation is to be applied to a series of arguments, the operation for the next argument can be started as soon as the privious one has finished. If the operation can be split into n suboperations there can be up to n of them in progress at the same time. At the beginning and at the end of an argument series only parts of the pipe are busy. Therefore the agument series should be as long as possible. Further overhead occurs if the execution times of the suboprations differ. In this case there must be some kind of queues between the stages of the pipe and methods to prevent that these queues overflow. The operations might be very elementary like floating point operations. In this case the suboperations might be realized in hardware leading to vector computers. But pipelining is also of interest in th form of macro pipelining where the suboperations are implemented by programs. An example is the simulation of telecommunication systems.

The principle of concurrency consists in extracting several independent instruction streams and processing them separately on different processors. In the most simple cases the same sequence of instructions is applied 3 to the elements of an argument set. Proceeding in this way is called data parallel computation. Carried to the extreme the same program is executed on several processors for different parts of the argument set. This special form is known as the Single Program Multiple Data (SPMD) approach.

Full use of these principles can only be made if application programs are implicitly or explicitly characterized by a sufficient amount of parallelism of the kind the computing system offers. That this is a nontrivial task may be seen by the following considerations.

[Slide 4]

The dynamic behavior of parallelized programs typically shows an alternation between serial and parallel regions. Normally the different threads in a parallel region don't have the same execution times. The threads may differ in their starting time, they may need coordination, or they may have different execution times. Another point is whether the number of threads equals the number of processors or whether at least all parallel regions comprise the same number of threads.

The computational potential of high-performmance computing systems is often characterized by summing up rough estimates of the performance of the individual units which might work in parallel. The thereby obtained peak performance limits the maximum possible performance, but does not say anything about the performance which can be reached in a special application.

[Slide 5]

The problems arising from using parallelism are impressively demonstrated by Amdahl's law. As already mentioned in the most simple cases parallelized programs will execute in a sequence of serial and ideally parallel sections. In these cases, according to Amdahl's Law one obtains the shown curves for the reachable efficiency defined as the portion of the peak performance used. The left figure shows the reachable efficiency depending on the fraction that can be parallelized and on the number of processors. The curve for n equal to 1000 demonstrates that in the case of massively parallel systems an acceptable efficiency can only be reached if nearly 100% of the work can be ideally parallelized. The figure on he right side shows how high the parallelized fraction must be to reach for example 75% efficiency dependent on the number of processors.

[Slide 6]

Another hint to the difficulty of using hardware parallelism in an efficient way is given by some results of the Perfect Benchmark suite. It consists of various scientific programs that are believed to be typical for some classes of applications. The results for some well known vector computers and multiprocessors show the variance of efficiency over different types of algorithms.

[Slide 7]

The problems to adapt programs to architecturally different computer systems may be seen by the following example. The selfconsistent Hartree-Fock method for calulating the electronic structure of molecules is in widespread use. In computations using this method a lot of time is used by calculating the incomplete gamma function for a very large set of arguments. For three methods approximating this function with the same accuracy runtimes are given in this figure. Remarkable is the fact, that without using vector operations the Taylor series approximation runs fastest whereas on vector computers the method of Frchtl and Otto is the best one. This result shows in an impressive manner that at least in some cases the algorithms have to be adapted to the type of computing system. Especially it demonstrates the limits of any kind of automatic adaptation.

From these few remarks it should be clear that structuring programs for parallel execution and thereby balancing the load of the different system components is a difficult task. On the other hand the spectrum of available systems varies rapidly. As a consequence, portability becomes a vital aspect for the production of software to be used over longer time periods or to be part of different applications or to be run on different hardware platforms. The most successful method to create highly portable software is to develop it for a fictitious virtual machine. This virtual machine has to be chosen in such a way that programs written for it can be automatically adapted to a broad spectrum of real machines. Widespread and successful models for serial programs are, for example, the programming model underlying Fortran, the object oriented programming model, and the database programming model underlying SQL.

[Slide 8]

In most of all cases programs written for a particular programming model are not immediately transformed to the computer model describing a special hardware system. Instead they are mapped onto an execution model given by the applications interface of an operating system. From the point of view of a single user his program should be mapped in such a way that it executes as fast as possible. In contrast, from the viewpoint of the owner of the computing system it is desirable to use all parts of the system most of the time. This lead to the development of operating systems multiplexing the hardware between several virtual machines. Timesharing systems are a classical example. Their execution model comprises several virtual sequential machines with a common file system, whereas the computing model of the hardware might be a monoprocessor with several disk subsystems. The figure tries to illustrate the connections between the three models mentioned, adapted to multiprocessors and distributed systems.

With respect to traditional computing systems, by using these ideas a high degree of portability has been reached without loosing an intolerable part of the hardware performance possible. In contrast, for high performance computing systems the situation is not nearly as good. During the last years some progress has been made, but for the development of highly efficient or portable programs it is still necessary to take into consideration all three of the aforementioned models. In order to give an overview of the state-of-the-art, it therefore seems best to first consider the most fundamental features of the relevant computer models. It follows a description of today's execution models and finally the state of relevant programming models will be discussed.

2 Architectural features of hardware

In programming high performance computing systems inevitably the architecture of the hardware has to be taken into account. But there is such a great variety, that prior to this some classifications of the most relevant features are needed. The vast majority of machines is based on the control flow principle, according to which instructions are executed one by one in a predefined order. Each program is viewed as a single stream of instructions. There are a few other interesting principles, like data flow, reduction, and the principle of neural networks. However, with respect to high performance computing none of those could until now demonstrate its superiority. Therefore, the presentation to follow will focus on architectures based on the principle of control flow.

[Slide 9]

The essential components of a classical monoprocessor are

- the processor, executing the instructions in the order given by a program,
- the primary memory containing program text and data, it is connected to the processor by a bidirectional data transport system,
- the peripherals controlled by the processor and connected to the primary memory. For high performance computing systems the most important classes of peripherals are disks representing 10 secondary memory and connections to communications networks.

Without changing the architectural concept higher performance is attainable only by increasing the clock frequency and the effective data transfer rates. It is a generally accepted opinion that in both directions some progress will still be made, but at a much slower pace than in the past. An obvious approach to increase performance in spite of these facts is through pipelining and/or concurrency.

2.1 Single instruction multiple data (SIMD) models

[Slide 10]

Possibilities for replication of parts internal to a processor are

- pipelining of instruction fetching, decoding, executing, and back writing,
- pipelining of functional units operating on sequences of operands,
- chaining of several functional units,
- replication of functional units.

Normally the first one does not affect programming. Combinations of the last three mechanisms are typical for vector computers. Because of the pipelining principle it is necessary to apply operations to long sequences of operands. In addition operations have to follow each other in such a way that most of the time all pipes can work in parallel. The increase of the performance of processing units has to be accompanied by higher rates of data transport between processor and primary memory.

[Slide 11]

Similar transport problems arise at several occasions and they have to be taken into consideration by programmers. Therefore I'll have a closer look on them. In all kinds of data transports there can be distinguished three phases:

- during the set-up phase the request is scheduled and transmitted and sending the data is prepared.
- during the transport phase the requested data is transmitted
- during close-down time the transport line is prepared for accepting new requests.

While the close-down time in most cases can be neglected, the set_up time, also called latency, is crucial for the efficiency and has to be taken into consideration by programmers because it lasts much longer than the transmission of a few bytes. Increasing the effective transport rate can be done

- by decreasing the time for scheduling requests
- by increasing the numbr of bytes demanded by a single request
- by using several independent transport lines
- by requesting the bytes long enough before they are needed
- by increasing the transfer rate.

All but the last point can be heavily influenced by skillful programming. In part state-of-the-art compilers can give support in reducing transport overhead.

Coming back to the problem of data transport between processors and primary storge the most relevant hardware mechanisms for increasing the efective transport rate are

- banking, which means dividing the memory into several partitions each with its own address and data path. In some systems (like MasPar) there is a fixed one to one correspondence between banks and processing elements.
- interleaving, which distributes the addresses cyclicly onto memory banks. If in such systems a program steps through vectors by an increment other than one, there is some danger in effect to use only a few banks and in such a way to dramatically reduce the effective transfer rate.
- caching, which inserts special high speed memory between the processor and primary memory.

To make optimal use of caching, programs should apply many elementary operations to a portion of data small enough to be stored in the cache. Therefore pipelining and caching are two contradictory principles. This is one reason why vector computers normally don't use caches.

[Slide 12]

The following examples of up to date SIMD systems demonstrate the great variety of these structures and the difficulty of taking full advantage of their computing power (performance data refer to solving linear equations and are taken from a lsit published by Dongarra). I have to remark that the list is far from being complete. Its purpose is only to give an impression of state-of-the-art SIMD systems.

2.2 Multiple instruction multiple data (MIMD) models

Another way for increasing the system performance is the replication of processors (including their ability to control the flow of instructions), of memory, and of peripherals. The variability of these systems reaches from single systems connected by some network, to multiprocessors.

In multi computer systems - as schematically shown by the figure - every computer executes its own program. Exchange of data between different computers can only be achieved by passing messages through a communications network. Because of this property systems structured this way are called NORMA (no remote memory access) systems. Typical systems of this class are the IBM SP2, the NEC Cenju-3, and the Intel Paragon.

[Slides 13 - 16]

NORMA systems can be used efficiently only if data may be distributed in such a way that only in rare cases a program accesses data not contained in its local memory. As Wilcke demonstrated in an interesting paper, there are many problems in the field of scientific high performance computing which can easily be adapted to NORMA systems. They compute very complicated functions of several arguments like the integral of the time-dependent Schroedinger equation or the simulation of physical systems with Monte Carlo methods for a large set of arguments.

[Slides 17 - 18]

At the other end of the spectrum are multiprocessors. These are characterized by a highly specialized network connecting all processors with all of primary memory. Ideally the time to transport data between a processor and primary memory is independent of the processor and of the memory addresses concerned. These structures are called UMA (uniform memory access) architectures. Connection networks with low latency and low probability for contention are realized until about 30 processors.

UMA architectures are the most common parallel machines, in part because most such machines are simply used as high throughput multiprogrammed, multi-user timesharing machines, rather than as execution vehicles for single, large-scale parallel programs.

But there are also algorithms that can be by far best parallelized on UMA architectures. An example can be found in fluid dynamics, where it is true that the numerically most efficient algorithms are inherently parallel, but they consist in computing a simple function on rapidly varying parts of the data. They perform best on UMA architectures, yet are very inefficient if transferred to NORMA architectures.

[Slides 19 - 22]

As a compromise between NORMA and UMA systems, there is a strong tendency towards so called NUMA (non-uniform memory access) systems. In these systems every processor has associated with it a local memory, but it can also be addressed from all the other processors - however,

at the price of a somewhat longer access time. They can be programmed like multiprocessors but with a degradation of performance if data cannot be held together in local memory. The degradation can be reduced by using caching techniques. Using caches with such systems raises the question of how to ensure consistency, if the same part of primary memory is cached in several caches. At the extreme ends there are systems like CONVEX SPP1 with automatic global cache consistency, and systems like CRAY T3D which leave the solution of these problems to the application software.

2.3 The I/O subsystem

[Slides 23 - 24]

When speaking about high performance computing people primarily think of computing power. But most large applications also need a large amount of secondary storage. This is especially true for commercial applications like large database systems. During the last years much progress related to capacity, reliability, and transfer rates of disk storage has been made. The most important architectural impovements within disk systems are the RAID systems, where RAID stands for redundant array of inexpensive disks. For improving throughput only RAID level 0 is of interest. In these systems multiple independent disks are organized into a large, high performance logical disk. Data are striped across multiple disks. By accessing them in parallel, high data transfer rates are achieved. But it does not reduce start-up times for single transfers. Usually that does not matter in scientific computing, where mostly large blocks are transferred.

[Slide 25]

The unwanted effects of start-up times can be reduced if at every instance there are several accesses to be served and several disk systems to be controlled by different processors. With this in mind DeWitt and Gray classify systems for database systems into shared disk multiprocessors and shared-nothing machines. For relational database systems they favor the latter form. It allows to easily port existing database systems by distributing relations among disks at different processors. By function shipping it allows for a high degree of parallelism when answering SQL queries. For example, Oracle has been implemented atop a 64-node nCUBE shared-nothing system, outperforming in far conventional mainframe systems. In the former terminology shared- nothing systems are NORMA systems with a full-fledged operating system running on each node. As already mentioned this architecture is also well suited to a lot of applications in computational physics. So this class will be one the important architectures that should be taken into account when developing new programs.

[Slides 26 - 27 give an overview over today's MIMD-architectures]

3 Operating systems and their execution models

[Slide 28]

Operating systems are serving a twofold purpose:

- For a single application they offer an execution model that is easier to handle than the computing model of the underlying hardware.

- For better use of the system they multiplex the hardware between different applications by offering so called virtual machines.

To programmers the architecture of these virtual machines constitutes the execution model. The most important criteria to judge new execution models for high performance computing are:

- simplicity: The concepts used by an execution model should have a clearly defined meaning, which can be described without recourse to hardware details.
- efficiency: The concepts should not introduce overhead that cannot be recognized on the level of the execution model. They must allow for using the hardware in an efficient way.
- portability: They should allow for formulating programs on a level of abstraction which automatically ensures adaptability to different computing models.
- compatibility: The concepts should be compatible with already approved ones being in widespread use.

[Slide 29]

With these criteria in mind certainly Unix offers itself as a basis for the development of appropriate operating systems. Unix has become a popular system in scientific computing since it is reasonable powerful. Because of its widespread use and with the mentioned criteria in mind certainly Unix offers itself as a basis for the development of appropriate operating systems. It brings with it the advantages of all mechanisms needed to multiplex processors, and to communicate between processes. The execution model of Unix is a NORMA architecture.

The most basic concepts of Unix are processes and files. Processes are abstractions of classical sequential monoprocessors with their primary memory, files are abstractions of peripherals, both I/O devices and secondary storage. Most important are files as persistent storage, i. e. storage existing unrelated to the lifetime of the process and principally accessible for all processes, this way being a kind of shareable storage.

It is common practice that execution models of the Unix type offer the possibility for executing several processes at a time. All these operating systems have mechanisms to allow for processes to interact. In particular these features play a vital role for execution models for systems with more than one processor. Wettstein differentiates between three basic forms of process interaction:

- coordination: This class of mechanisms allows to coordinate the execution of different processes in time. Well known mechanisms are Dijkstra's semaphore operations and, with respect to parallel programming, the so called barrier synchronization.
- communication: Most common mechanisms of this class are message-passing mechanisms. They differ widely in details, like in addressing receivers and the coordination of send and receive operations.
- cooperation: This class subsumes all mechanisms based on the use of virtual or real shared storage. The most common are using global files.

All three classes can be efficiently implemented for UMA and NUMA computing models, the same way they are implemented in traditional operating systems. By comparison, in NORMA architectures cooperation is achievable only with substantial overhead. Therefore there is a strong tendency not to include cooperating mechanisms into execution models for NORMA architectures, with the exception of global files. Additionally for these architectures processes and files offer themselfes as units of distribution.

But there are also several disadvantages. In systems specialized for scientific computing, there is often a need to equip several of the processors with the ability to control secondary storage. But the file system constitutes a large part of Unix. To avoid this overhead at every processor, mostly two versions are used: The full version on a few processors serving as hosts of the system, and a strongly reduced version on the other nodes. Yet the file system is so intimately integrated into the Unix system that indeed two different kinds of an operating system are needed.

A severe drawback of Unix becomes apparent by the following observation: Between two points of coordination, some applications apply only a few instructions in parallel to all components of a vector. To use processes for parallelizing this type of computation is inappropriate because of the high overhead in creating, blocking and deblocking processes.

[Slide 30]

A much better solution is possible by using modern microkernels like Mach. Such a microkernel can be adapted to special needs by adding a suitable server. If needed, this can be done in dynamical fashion. Mach for example can be supplemented by an Unix server including a local file system. In this case all processors run the same microkernel, creating in this sense a homogenous system. Where necessary, a Unix server can be added. Mach is based on five basic concepts: tasks, threads, ports, messages, memory objects.

Most important is the distinction between tasks defining an execution environment only and threads which are roughly independent program counters operating within a task. A task plus a single thread gives a traditional process. A task with several threads somwhat like anvirtual multiprocessor. For UMA and NUMA architectures it is a more natural concepton than the Unix execution model. Communication is based on ports and messages. Messages are communication end points assigned to a task. Messages are kernel-protected objects, that can be send or received through ports. A memory object is a virtual backing storage. By treating all primary storage as a cache for memory objects Mach implemnts a single level store. Above the Mach kernel more comlex features can be supplied through user level servers. Examples are Unix servers and file servers. Thus by establishing a microkernel on every processor they can be easily adapted to special needs. The execution model of Mach can be described as a NORMA architecture with multiprocessors as nodes. The model does not determine the structure of the I/O system.

Let me say a few words about the second task operating systems have to fulfill. Multiplexing of hardware was introduced to make better overall use of the hardware. The advantages of timesharing made up the resulting overhead. But with respect to high performance computing the situation is quite different. So the majority of operating systems in this area don't support timesharing. Instead they organize spacesharing by allocating to a task at starting time the proccessors and other rssources it needs during its lifetime. By this way there can be several tasks in execution at the same time. There are some further ideas to make better use of the hardware, for example through timesharing combined with gang scheduling, but these concepts are still far from beeing standard. In this respect operating systems are still in an early phase of development.

4 Languages and their programming models

[Slide 31]

Usually programming models underlying common programming languages are single threaded. They have to be extended in some way for use with SIMD or MIMD execution models. The criteria programming models hae to fulfill differ omewhat frm those for operating systems. The most important are:

- compatibility: means that new programming models have to consider the enormous amount of existing software. Moreover, there is a vast amount of knowledge in programming sequential machines which should not be lost. The most important programming models to be taken into consideration are those underlying Fortran, C and SQL.
- conceptual clarity: says that the number of basic concepts and operations must be not too large. They should be orthogonal, i. e. be freely combinable.
- portability: requies tht the concepts should allow for the efficient implementation into existing and expected architectures.
- efficiency: requires that if needed, also special features of a system have to be accessible.

In the near future, on behalf of compatibility and portability new programming models must present themselves as extensions of the Fortran model or, to a lesser extent, of the C model. On the long run there may be a tendency towards object oriented models as represented by C++. Since there are considerable differences between models suited for vector computers and those suited for MIMD systems, both will be discussed separately.

4.1 Programming models for vector architectures

[Slide 32]

Programming models for vector computers have reached a high level. They adhere to the execution model of (virtual) monoprocessors with special instructions for manipulating vectors. The concept of vector operation is unknown to Fortran 77, but there are substitutes which can be handled easily. Using simple enough loop structures, compilers can find out that in fact they are describing vector operations. During compilation the program can then be transformed into an intermediate form using vector operations. On this basis they further try by suitably permuting the sequence of vector operations to use all the pipes of the system in parallel. These transformations, however, are limited to loops with a rather simple structure. In some cases compilers allow further hints to vectorizable loops. But these directives are compiler specific and hindering portability. As a consequence, Fortran 90 comprises vector expressions. In the long term it seems interesting to pass over to object oriented languages like C++. These allow for the definition of new data structures and operations on them, allowing programs to be adapted in an evolutionary way to new hardwired data structures and operations. It also gives the possibility of choice. For example in some application it may be meaningful to define Hermitian matrices as a special data type. The operations on such matrices are then apparent to the compiler and can be tailored to the hardware in question without having any influence on the rest of the program. The concept of module in Fortran 90 is a step in this direction, but a very small one as compared to C++.

4.2 Programming models for MIMD-architectures

[Slide 33 - 36]

Programs to be run on top of execution models for MIMD architectures have to provide, for the most of the time, a sufficient number of threads (as demonstrated by Amdahl's Law).

The main stream of the development bifurcates into two directions. The first one primarily aims at NORMA execution models. The second one tries through creating new abstractions to describe programs for NORMA, NUMA, and UMA architectures in a unified way, leaving the adaptation to compilers and operating systems.

An obvious way for NORMA architectures is to choose some well known programming model for sequential machines and to extend it by a communications and distributing library. Well known libraries of this kind are PVM 3 MPI. They are nowadays available on a great many of platforms. Programs based on Fortran or C together with PVM 3 are highly portable. Benefits of these systems are their compatibility with the Fortran programming model and their availability. One of their disadvantages is their inability to deliver the full range of possibilities inherent to NUMA and UMA architectures to the level of applications.

The second approach needs new abstractions. A first step is, as already mentioned, Fortran 90 offering vector operations. Based on this abstraction efficient code can be compiled for UMA and NUMA architectures. In many cases it may also be compiled for NORMA architectures. A much more elaborated concept underlies HPF. Its programming model is that of a UMA architecture. But there are also abstractions to define a kind of affinity between, for example, components of vectors and threads. If the threads show a high degree of locality they can be compiled for NORMA execution models or for NUMA execution models, which themselves may use as a base hardware with a NORMA computing model (these systems are often called virtual shared memory systems).

This table gives an overview over the models used by the systems, that where presented in the former. Included are peak performance data, efficiency with compiled LINPACK solving linear equations, and best efficiency reached for solving linear equations.

5 Conclusions

[Slide 37 - 38]

Whereas in commercial computing the difficulties of programming high performance systems mainly is a problem of the suppliers of database systems, the situation encountered in scientific computing is quite another one.

In spite of intensive research during the last decade at least, programming for systems with parallelism of some or another kind is still a difficult task. The state of the art in programming parallel systems is appropriately described by Polychronopoulos. He writes `Presently the state of the art necessitates parallel programming on a case by case (i. e. application) basis.'

At present there is a strong tendency towards NUMA and NORMA architectures, both real and virtual. The fact that HPF aims at a programming model for this kind of systems is a further hint that in developing new programs for high performance computing one should have in mind these

kind of architectures. The main tasks are to make the potential parallelism obvious, to partition data in such a way that most of the time threads are working with local data, and to organize data transmission in a way that start up time can overlap with further activity of threads. There are no general methods to achieve these goals, and perhaps will never be.

References

[1] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings AFIPS 1967 Spring Joint Computer Conference, volume 30, pages 483-485, April 1967.

[2] DeWitt, D. and Gray, J. Parallel database systems: The future of high performance database systems. Communications of the ACM, 35(6):85-98, 1992.

[3] Dongarra, J. J. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS - 89 - 85, Computer Science Department, Univ. of Tennesee, Knoxville, Ten. and Oak Ridge National Laboratory Oak Ridge, Ten., Jan. 7 1995.

[4] Geist, A.; Beguelin, A.; Dongarra, J.; Jiang, W.; Manchek, R.; Sunderam, V. PVM 3 User's Guide and Reference Manual. Technical Report, Oak Ridge National Laboratory, Oak Ridge, Ten., May 1993.

[5] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0. Scientific Programming, 2(1-2):1-170, Spring and Summer 1993.

[6] Message Passing Interface Forum. MPI: A Message- Passing Interface Standard. International Journal of 34 S1ercomputer Applications and High Performance Computing, Special Issue, 8(3/4):159-416, 1994.

[7] Polychronopoulos, C. D. Parallel programming issues. International Journal of High Speed Computing, 5(3):413-473, 1993.

[8] Reid, J. The Advantages of Fortran 90. Computing, 48:219-238, 1992.

[9] Wettstein, H. Systemarchitektur. Hanser Studienbuecher der Informatik. Carl Hanser Verlag, 1993.

[10] Wilcke, W. MIMD computers for scientific applications. International Journal of High Speed Computing, 5(3):403-411, 1993.