2

A PROGRAMMERS VIEW OF PARALLEL COMPUTERS

Tor Sørevik

Parallab, University of Bergen

1 INTRODUCTION

As a programmer one can't worry about all sort of technical details of a computer. They are complicated and subject to fast and unpredictable changes. Instead one relates to an abstract model of the computer. For the conventional sequential machine the standard model, the socalled von Neuman-machine [GvN63], has proved to be remarkable useful and robust to the enormous technical advances which have been made in computer design. The main simplifications of reality in the von-Neuman model are that memory has infinite size, and that access to memory is free of cost.

The standard way of assessing the efficiency of an algorithm running on a von Neumann machine is by operation counts. If memory access was free of cost a careful operation count would be an accurate prediction of the computational cost, but unfortunately the assumption of free memory access does not hold in practice. Over the last years, we have experienced a much faster increase in processor speed than in memory bandwidth. Thus it becomes increasingly important to take the memory access pattern into account, when evaluating the efficiency of a program. As a result, the von-Neuman model works well for developing the logic in the program, but has limitations when predicting the real performance of applications, where memory access time is the bottleneck.

Moving to parallel computers the von-Neuman model is replaced by a PRAM model (Parallel Random Access Memory). Here the von-Neuman assumption of infinite sized memory and free access time is extended with the assumption that P processor might work simultaneously, and that they all have access to all the data. (Different versions of the PRAM model exists depending on whether or not to allow for Concurrent or Exclusive read and/or write).

When implementing a parallel algorithm developed for a PRAM model one has to take into account memory organization. Not only because this is an important variable determining the efficiency of the program, but also because most parallel programming environment insist on the programmers cooperation in memory management. To do this the programmer need to have, not only intimate knowledge of his application, but also knowledge on the memory organization of the parallel computer at hand.

Our attitude is the user's. High performance is what we want, not parallelism! Parallel computing is just an unavoidable result of our desperate desire for high performance. When looking at High Performance Computers, we therefore focus on the performance bottleneck. Today that appears to be fast access of data. Consequently we will in this paper discuss the memory organization and the interconnecting network of high performance computers.

In section 2 we discuss the memory hierarchy of modern computers and focus on how this influence the efficiency of the computation. We take the view here that multiprocessors typically introduce a new level in memory hierarchy, Local and remote memory. We describe the difference between shared an distributed memory. In particular we look at how this bisection of computer architecture is reflected in the programmability and the scalability of the computer. In section 3 we take a look a network with emphasis on cost and scalability. In section 4 we look to the future and make a few predictions on trends.

2 THE MEMORY HIERARCHY

Physical and economical laws have imposed a hierarchy of memory. [KS96], [HP90] The closer to the CPU the memory modules reside, the more expensive it is and the less space it occupy. Thus at the top of the hierarchy we find small and fast memory modules, called caches. The fastest caches are on the same chip as the CPU itself. There are physical limits to the sizes of on-chip cache, but as the packaging density increases there are room for transistors on each chip and some of these will be used to increase on-chip cache. Almost all scientific interesting problems require huge amount of data and we are soon forced to store data off-chip. Sufficient sized memory is than achieved by adding larger, but slower memory units, further away from the CPU. The next level are

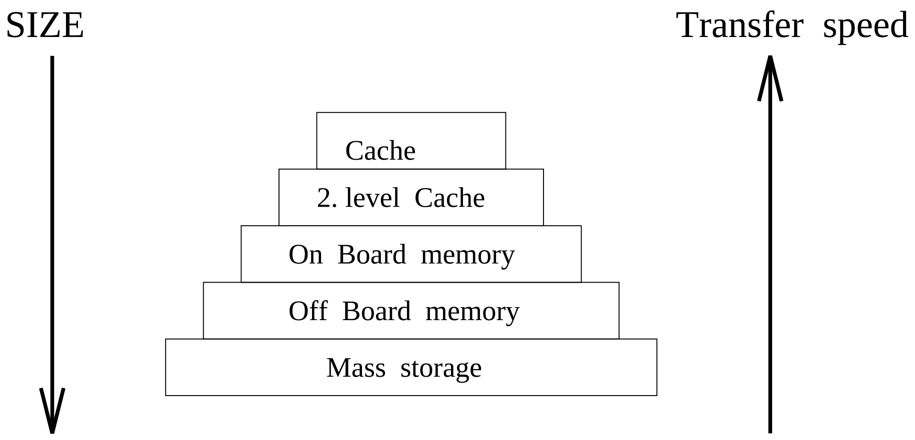


Figure 1 *The Memory Hierarchy of High Performance Computers*

usually SRAM-modules (Static RAM). This is faster, but more expensive than DRAM (Dynamic RAM) which the main memory usually is built of. Economy is usually the main limit for SRAM, level 2 cache.

The numbers of levels in the hierarchy displayed in Figure 1 is different from system to system. As is the size and speed of the data transfer time to the individual modules.

2.1 Cache effects

To illustrate the effect of having data in cache we have coded a simple `daxpy`-operation in FORTRAN and run it on different systems with various vector lengths. As is evident from Figure 1 the speed of this operation behaves like a staircase function where the steps coincide with the the size of the caches on different system (See table 2).

Efficient use of caches is not only a pure hardware question. Equally important is software. In particular compilers and how well they are able to understand your code and do optimization tricks as: pre-fetching, loop-unrolling, branch prediction, out-order-execution etc. This, in turn, do depend on the interplay between processor design, compiler/language design and the programming style of the individual programmer. To indicate the effect of this, we implemented the same simple `daxpy` operation in various flavor of F90 and run in the DEC

	clock cycle	flops/cycle	peak perf.	L1 cache	L2 cache
RS580	62 Mhz	2	124 Mflops	64 Kb	-
RS590	66 Mhz	4	264 Mflops	256 Kb	-
AXP 21064	233 Mhz	1	233 Mflops	16 Kb	1 Mb
Sparc 10	60 Mhz	1	60 Mflops	16 Kb	1 Mb

Figure 2 *Technical specification of the 4 RISC-processor used in experiments reported in Figure 1*

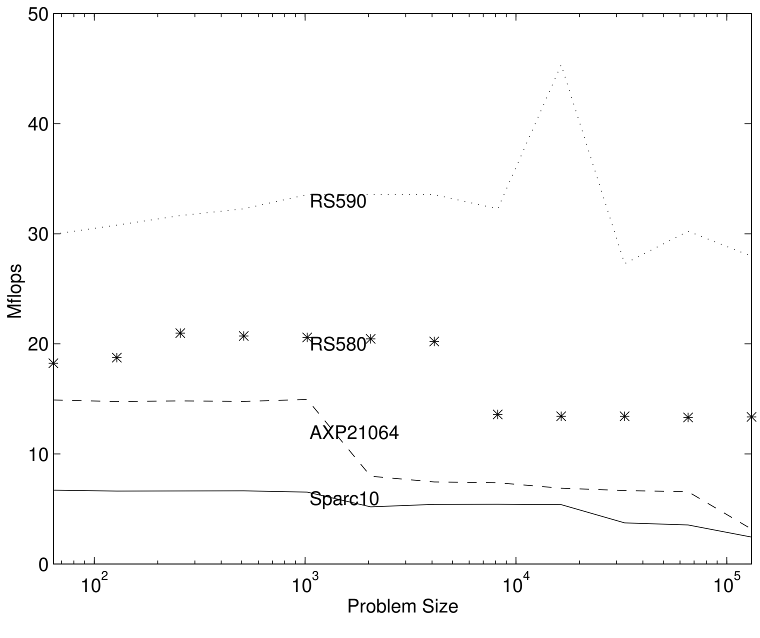


Figure 3 *Performance of the daxpy operation on 4 different RISC processors, compiled with standard FORTRAN compilers*

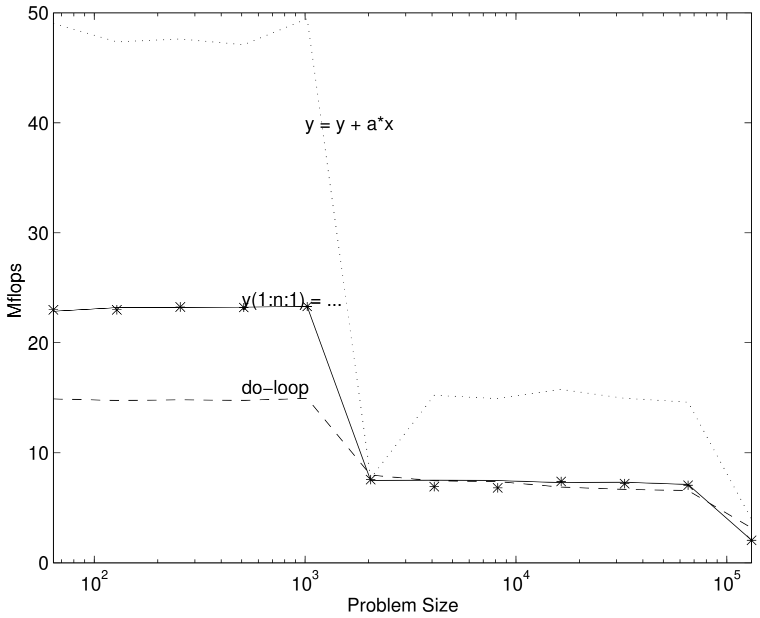


Figure 4 *Performance of the daxpy operation on the DEC alpha platform using different syntax and compilers*

alpha, which has a mature F90-compiler. The results are displayed in Figure 4.

2.2 Off-board memory

In Figure 1, what is called, the "off-board memory" is the only part that is usually found in parallel systems and not in sequential ones. Thus in the view of memory hierarchy parallel system may at first glance not introduce any new concepts. However, a closer look shows that (at least) two new problems are introduced. These are related to the fact that off-board memory usually are divided in separate memory modules, to which the individual processors have Non-Uniform Access time (so-called NUMA architecture), implying the following problems:

1. *Which memory module to store the different data in and when to rearrange the data between the modules, such that access time are minimized?*

2. *How to solve conflicts when many processors are writing and reading the same data?*

Good answers to these questions are hard to find without explicit knowledge of the problem at hand. Therefore attempts to construct general solutions, which could be implemented, at the system level have had limited success. Consequently on most parallel system the programmer himself is responsible for taking care of data layout, data movement and sorting out the access pattern for all processors to the individual data segments.

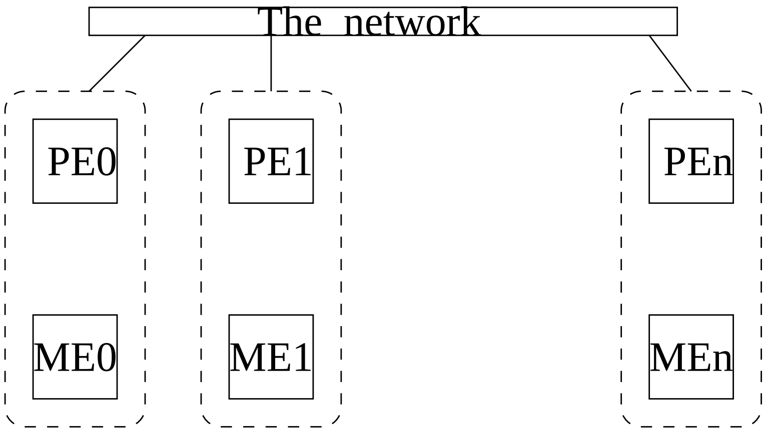
Shared Memory Processors

In a shared memory model all the processors have the same access to all the data. This gives greater scope for the system to handle the two problems we describe above. Doing this is however not without problems. The processors of all the popular system of today is cache-based RISC processor, and the caches are local to the individual processors. Thus, unless care is taken, if a processor fetch data from the shared memory and this data already has been copied into another processor cache, we have no guarantee that these data not have been updated in the first processor's cache. Thus it is necessary to implement some mechanism for keep each other updated on where the most reason version of the data reside. This problem is known as the cache coherency problem. Various techniques is applied for solving this, depending on the interconnecting network [DCS88], [Ste90].

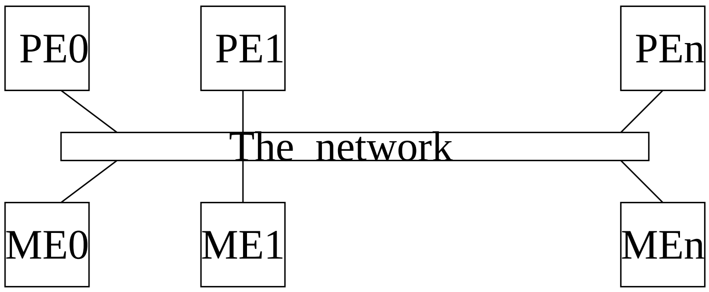
Buses offer a relatively simple an inexpensive way of connecting multiple processors to the shared memory. The coherency strategy used on busses is called *bus-snooping*. When a processor update variables in its cache, it issues a *consistency command*. The other processors are "snooping" on the bus and mark their copy, if they have any, as invalid. Most multi-processor high-end workstation implement their shared memory using busses. The most prominent of these is SGI's Power Challenge and Digital's alpha-server.

The limitation of bus-based systems is their scalability. The bus has a fixed bandwidth which has to be shared between the processors. Thus increasing the number of processors implies less bandwidth to each, the bus becomes saturated

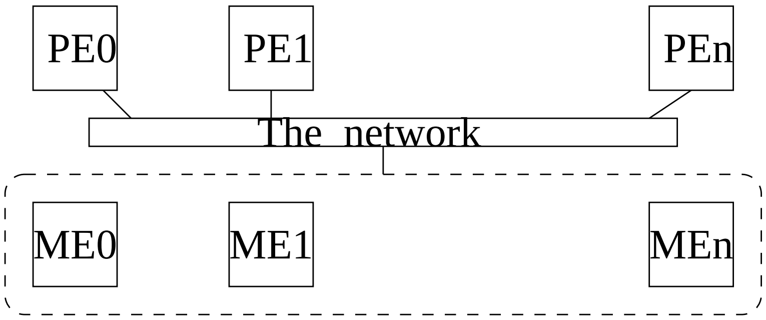
Distributed Memory System



Distributed Shared Memory



Shared Memory System



and the processors spend unacceptable long time waiting for data to arrive. The snooping techniques, just described, adds extra load to the bus traffic.

The limited scalability of buses has lead to the construction of *Distributed Shared Memory* systems. In this case the memory too is distributed into different modules. Supplying this with a scalable interconnecting network which allows simultaneously access to the different memory modules, by the different processors should in theory give a scalable system that maintain the look and programmability of a shared memory system. The DASH-project at Stanford has explored this opportunity [LLW+92]. They have developed a directory based cache-coherency system for this kind of architecture. The idea is that each memory module keep a record of which of the processors that have a copy of the different cachelines. Information on when and where cashes are updated are recorded here. Convex has implemented this system in their SPP series using a crossbar switch as their interconnecting network.

Distributed Memory System

On a Distributed memory computer each memory module is local to a specific processor or a "node". With no shared data we avoid the cache-coherence problem, instead when processors need data residing on another node, data have to be exchanged through explicitly sending the data across the network. Most system leaves the job of finding out which data to send and where to send them to the programmer. Having to cope with these low level issues the programming of distributed memory computers becomes a a complicated task. On the other hand provided a fast interconnecting network is available these systems have the potential of scaling well. They also are reasonable simple to build. In principal one can just connect together standard workstations. From a conceptual point of view the difference between a DMS parallel system and a cluster of networked workstation is that the later usually have a slower network, and lack software that permits a *single-machine* image of the system.

Popular system in this category is IBM's SP1 and SP2, Cray's T3D and T3E and the Intel Paragon.

Hybrid systems where each node in a distributed memory system is a shared memory processor system do also exists. Examples are the latest version of the Intel Paragon and the Parsytec GC. On these system there is 2 bus-connected processor for computation in addition to communication processors on each node.

3 COMMUNICATION NETWORK

3.1 Scalability

If an algorithm for a fixed sized problem has a fraction f of the computation which only runs in sequential mode, than the speed-up of the computation is bounded by $1/f$, regardless of how many processors one apply. This observation is usually called "Amdahl's Law" [Amd67] and has for 3 decades been hold as a strong argument for the limitations of parallel computing. Although appealing in its simplicity, this argument does not capture the entire truth. For most application the main motivation for going parallel, is being able to solve larger problems, and in most of these cases we find f decreasing while increasing the problem size [Gus88]. This observation has lead to a strong interest in what is called scalable algorithms. That is, keeping the efficiency constant, when at the same time increasing the number of processor and the problem size. For a discussion of scalability and how to measure it see [GGK96].

To get scalable performance of a scalable algorithm one needs a scalable architecture. When the number of processors increase, we must make sure the hardware is able to shuffle the data sufficiently fast. On the bus-connected shared memory system, the total bandwidth is constant. Thus when adding more processor less bandwidth becomes available per processor, and it soon becomes hard to obtain increased performance when adding more processors as the memory bandwidth become a bottleneck. For this reason these system does not scale well.

Whether or not a Distributed (Shared) memory system scales well depends on the interconnecting network, and that is the issue for this section. First we describe some network topology and discuss for which category of communication they work well, and than we discuss some of the popular hardware.

3.2 Topology

We may distinguish between 2 main categories of communication: 1) Local communication; where each processor only talks to a few neighbors, and 2) Global communication; Here we particular have mind collective operations where all processors take part. For local communication a mesh is the topology of choice. (See fig 3.2). This is a cheap network in the sense that for $O(P)$ processor only $O(P)$ channels are needed. But even so, if the communication is only to nearest neighbor, each processor can send and receive data at the same speed regardless

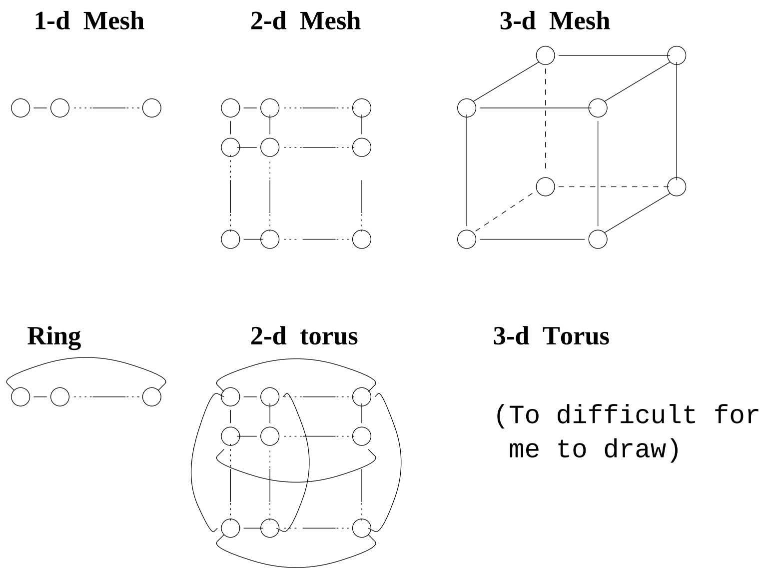


Figure 5 *Meshes of different dimension, with or without wraparound*

of how many they are. On the other hand, the diameter of the network (the farest distance between any pair of nodes) increases with the number of processor, making global communication more and more expensive, when increasing the size of the network.

By global communication we mean operations like broadcast, where data on one node are being sent to all the other nodes, or reduction operations like a global sum of data spread across all the processors. A simple and efficient network for broadcasting is a star. A binary tree is what's needed for doing the parallel prefix summation. In a star-network the node in the middle becomes a serious bottleneck for any other communication than the broadcast. While a binary tree also works reasonably well for broadcast. $O(logP)$ operations as compared to $O(P^{1/d})$ for a d-dimensional mesh.

Many applications need different kind of communication. Thus one might like to use a binary tree for some part of the computation and apply a mesh to other parts. One answer to this problem is to use a more advanced network which enable you to emulated the basic ones. The most popular of these is the hypercube network, which has been used in a number of commercial products. An other possibility is the shuffle-exchange network or the fat-tree.

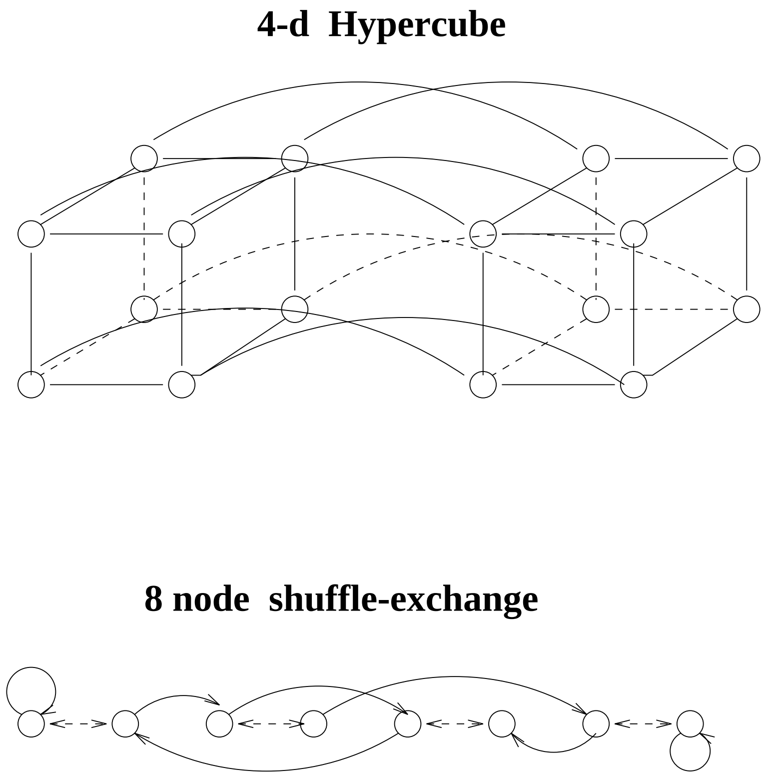


Figure 6 *Hypercube and shuffle- exchange network*

The following table summarize some of the key parameters for the topologies briefly discussed here.

network	no. of channels	diameter	bisectional bandwidth
1-d mesh	$P-1$	$P-1$	1
2-d mesh	$2(P-\sqrt(P))$	$2(\sqrt(P)-1)$	$\sqrt(P)$
star	$P-1$	2	1
binary tree	$P-1$	$2(\log P-1)$	1
hypercube	$P \log P$	$\log P$	$P/2$

3.3 Network Hardware

There are 2 basic strategies for sending data in a network *store-and-forward* routing and *wormhole routing*. In the case of store-and-forward a packet is completely store in one node before it is transmitted to the next. This makes the completion time approximately proportional to the number of hops it travels. In wormhole routing the head of the packet is advanced directly from the incoming to the outgoing channel. Only a few control digits are buffered at each node. With the appearance of wormhole routing, the communication time no longer is proportional to the distance. With wormhole routing it becomes possible to do efficient global communication on meshes. Note however that in wormhole routing the data might be spread across many channels between the source and destination node, and consequently more than one data set might like to traverse the same edge in the network simultaneously. Thus unless there is sufficient bandwidth we might experience severe contention on such a network.

The Intel Paragon uses a 2-d mesh with wormhole routing while the Cray T3D and T3E apply a 3-d mesh with wormhole routing.

Crossbar switches provide a contention free and fast interconnect between n processors. It does however require n^2 crosspoints. For this reason it becomes expensive to build large crossbars. Instead to build an affordable and large switching network one arranges these in hierarchy or multistage switches. Systems with this type of network are IBM's SP1 and SP2, Convex/HP's Exemplar SPP-series. Fujitsu's VPPxxx systems have only stage switches.

4 FUTURE TRENDS

The technological development gives possibilities and sets limits for the next generation supercomputers. The marked determines which of the possibilities the vendors will explore. So let us first make two observation concerning the marked:

1. The volume of the traditional supercomputing marked is increasing at a much slower speed than the rest of the computing marked.
2. The profit is small in this marked segment.

As a consequence of point 2, the vendors are forced to cut costs, and since product development is the major investment in this branch of the computing business, one can not longer afford to tailor every single component of a supercomputer. Point 1 implies that less emphasis is put on the high-end scientific marked in the development of the next generation computers. These trends are acknowledged by all the major vendors. Since the last couple of years, they have all stressed that will try leverage on commodity parts in their future products. This trend is most pronounced for the processor and memory system. While proprietary vector processors with associated memory system used to be the processors of choice for supercomputers, we nowadays find that about 4/5 of the computers listed among the top 500 supercomputers are based on commodity RISC processors. One of most successful Parallel system over the last couple of years is the IBM SP1 & 2. The nodes of an SP2 is essential a RS6000/590 workstation. Not only the processor board with CPU and the memory system, but also disk and I/O-system. Moreover each node is running the full AIX operating system. The only special hardware is the interconnect, a fast multistage switch.

This trend might take some of the glamour away from supercomputing. A supercomputer is not anymore, in every way, the apex of the high-tech computer pyramid. One might argue that it is just another standard high-end workstation, only more powerful. Apart from "the-loss-of-the-glory", this is probably a good trend for the users. He will find a more continuous path from the development environment on his workstation to the high-end production platform, where he runs his grand challenge applications. He also will have a higher probability of being able to afford the new system. The development, just described, are already taking place. The need for vendors to provide a continuous path from workstations to high-end supercomputer, has had great impact in the industry. The small, independent start-up companies in the supercomputer business have either gone out of business or have been bought up by "the big five" workstation producers.

Using commodity components means that vendors can concentrate their effort on solving the issues specific for parallel computing. Such as: Low latency, high bandwidth interconnect, cache coherency on shared memory system, and all flavors of software tools for parallel computers.

REFERENCES

[Amd67] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, Washington D.C., 1967. Thompson Books.

[DCS88] M. Dubois and F. A. Briggs C. Scheurich. Synchronization, coherence and event ordering in multiprocessors. *Computer*, 21(2):9–21, 1988.

[GGK96] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency function: A scalability metric for parallel algorithms and architectures. *IEEE Transactions on Parallel and Distributed Computing*, To appear, 1996.

[Gus88] J. L. Gustavson. Reevaluating amdahl's law. *Communication of the ACM*, 31(5):532–533, 1988.

[GvN63] H. Goldstine and J. von Neumann. *On the principle of large-scale computing*, volume 5. Pergamon, 1963.

[HP90] J. L. Hennesey and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, Calefornia, 1990.

[KS96] L. I. Kontothanassis and M. L. Scott. *Memory Models*, chapter 24, pages 699–722. McGraw-Hill, New York, 1996.

[LLW+92] D. Lenoski, J. Laudon, W.-D. Weber, J. Hennesey, M. Horowitz, and M. Lam. The stanford dash multiprocessor. *Computer*, 25(3):64–79, 1992.

[Ste90] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.