Tightly Coupled Multiprocessing: Architectural and Technological Challenges

Abstract

Due to a number of fundamental hindrances, parallel computing has not yet become a practical general purpose technology comparable to current serial computing technology. Overcoming these hindrances is a major challenge that information society faces in the 21st century.

We advocate and explore a solution that departs from the strong reliance on the locality of data with respect to the individual processor. Under our approach, the multiprocessor is analogous, in fact, to a uniprocessor: The collection of processors acts as a single "super processor", working under fine or medium granularity vis-a-vis a symmetric shared memory. An architectural/physical model of such a system is outlined, and a simple estimate for the clock frequency is given. The main novel part of the proposed "super processor" is a *high flow-rate synchronizer/scheduler*, which coordinates the parallel work. The macro and micro architecture of the synchronizer/scheduler are described. The proposed solution points to secondary challenges, of a more technological nature.

Keywords and Phrases: High performance computing, symmetric multiprocessing, tight coupling, synchronizer/scheduler.

1. The Challenge

1.1. Problem Definition

Massively parallel MIMD systems have three fundamental problems:

1. The memory latency (or communication latency) problem.
2. The software engineering problem.
3. The problem of coordinating the parallel work.

The primary cause of the memory latency problem in multiprocessors (or the communication latency problem in multicomputers) is that the establishing of a path and the transferance of a datum between remote components are slow, relatively to the propagation time of signals within a processor. The secondary reason is the competition over interconnection resources. The memory latency problem also spawns the cache coherence problems, when caching is pursued as a solution. Next, the software engineering problem is the difficulty of producing effective parallel code, in comparison to the relative ease of producing serial code. Finally, the problem of coordinating the parallel work is the problem of synchronizing between the instruction streams in a flexible and efficient way, without undue delay in issuing ready to exectute work, and of maintaining efficient scheduling and keeping load babancing. These three problems may hinder

the fulfillment of promises stated in terms of peak rate.

Tight coupling, in the original sense of the very usage of shared memory, has emerged as an answer to the software engineering problem: The merit of the programming model based on shared memory, relatively to the message passing model, is in being more similar to a uniprocessor's programming model. The concept of tight coupling can also be attributed a connotation of working under fine (or at least medium) granularity: When the granularity is finer, the accessing of data tends to be less localized, and the volume of activity needed for coordinating the parallel work increases; the processors work in tighter cooperation with each other and behave as a single large body, a "super processor", vis-a-vis the shared memory. This must be expressed both in the hardware and in the programming model. The motivation for working under finer granularity lies in the need to extract more parallelism out of an algorithm. Yet so far, fine granularity has become a commercial reality only under very small scale parallelism—instruction level parallelism in superscalar uniprocessors.

The original problem that we would like to address is the following one: *Find an integrated remedy for the three fundamental problems listed at the beginning of this paragraph, under a setting of tight coupling (in its latter sense).* Section 2, which occupies the larger part of this paper, outlines a non-conventional solution approach for this problem. Yet this solution may point on a secondary problem, of a more technological nature, associated with the density of packaging of circuitry at the scale of a large system, which might emerge as a major technological challenge in the 21st century.

1.2. The Significance of the Problem

We would like to cite the words of David J. Kuck, who has discussed the state of the high performance computing field in his 1996 book [4]. They are still valid today. From Kuck's words it follows that the problem posed at the the end of previous paragraph pertains to a major open challenge on the agenda of society. According to Kuck, turning parallel computing into a practical general purpose technology is a critical necessity that has not yet materialized:

> The advent of commercially available parallel computer systems in the 1980s must be regarded as a major milestone in the history of computing. However, despite the great strides made in parallel processing in the past 20 years, the technology still has a long way to go before practical parallelism emerges (p. 38).
>
> ...This sequential speed squeeze may have far-reaching effects in the next century. ...A parallel processing imperative arises from this sequential speed squeeze...(p. 41).

1.3. Comments on the State of the Art

We would like to comment about three characteristics of the state of the art:

a. *The lack of orientation of current supercomputers towards fine granularity.* Information about current supercomputers can be found at the "top500" web site [6]. Current supercomputers are not oriented towards fine granularity in two respects. Firstly, the access to a remote component is slow, relatively to propagation times within a processor (although the degree of slowness varies widely) Secondly, the coordination of the parallel work is carried out by the processors themselves in software, by operating synchronization or communication primitives alongside other instructions. This may entail too much overhead when working under fine granularity. In addition, "hot spots" and bottlenecks may be incurred, especially when dynamic load balancing is practiced. Hence, these computers do not embody tight coupling in the sense of Paragraph 1.1. Moreover, there is a tendency of convergence between the shared memory model and the message passing model—see, for example, Culler and Singh (1999) [2], or Protić et.al. (1998) [5].

b. *The remaining of dataflow architecture outside of mainstream.* The dataflow concept embodies tight coupling in the sense of Paragraph 1.1, to a certain extent,by being oriented towards fine granularity, and by the fact that the collection of processors is viewed by the programmer as a single body that the programmer need not enter its details. However, dataflow architectures have been criticized, as far as their efficiency is concerned.

c. *The flourishing of cheap PC clusters.* A cluster of PC's or of workstations connected through a few off-the-shelf communication switches forms a cheap substitute for a supercomputer. And since current supercomputers are not oriented to tight coupling (in the sense of Paragraph 1.1) anyway, why not content with a substitute that differs only quantitatively?

2. Our Proposed Solution

This section presents a multiprocessor architecture aimed at addressing the challenge described in the previous section. This is just yet another solution joining the many solutions already proposed, but we maintain that it is legitimate to suggest ideas as long as the challenge remains open.

2.1. The Overall Architecture

We choose to start the description of our solution from the architectural/physical model depicted in Figure 1. The system comprises three elements: The "super processor", a

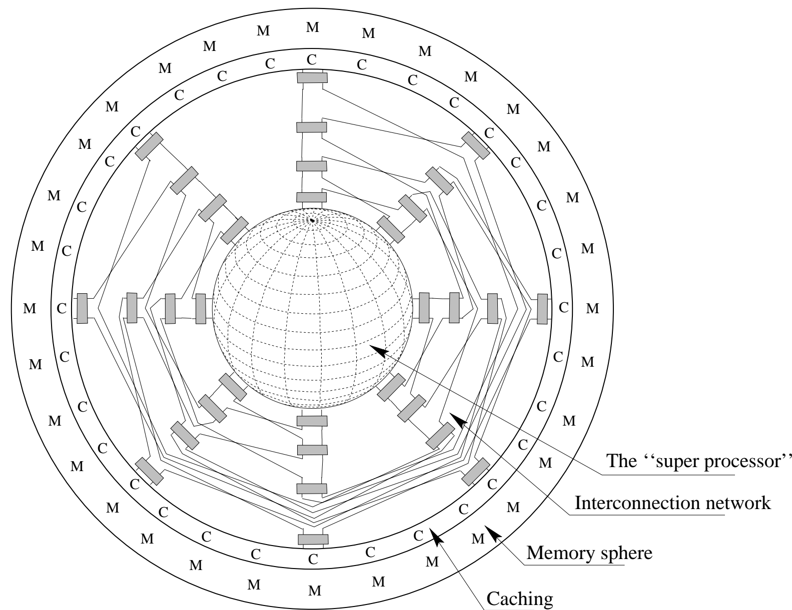

Figure 1: An architectural/physical model of the overall system. The 16×16 baseline interconnection network in the figure illustrates the possibility of bounding the overall length of the wiring between any two points by the circumference of the outer circle, at least in the one-dimensional case.

global symmetric shared memory, and an interconnection network. The caching of data is done at the memory side of the network, whereas the caching of instructions is done at the processor side. (Caching of data at the side of the processor is possible as well, with the coherency being preserved as a by-product of the programming conventions, but the orientation towards tight coupling just leads to caching of data at the memory side). Input/output pathes are omitted from this description.

Memory
references are emitted from $N_{\text{prcessors}}$ different points on the spherical surface of the "super processor"; they are generated by the same number of individual processors. These references reach $N_{\text{prcessors}} \cdot M_{\text{interleaving}}$ different points on the memory sphere via the interconnection network; the parameter $M_{\text{interleaving}}$ is aimed at controlling the probability of collision in the interconnection network, and is not related to memory interleaving in the classical sense. From the description of the "super processor" later in Paragraph 2.2 it will follow that the traffic between it and the memory does <u>not</u> contain references to synchronization data or to any other type of "hot spots". We propose to build the system as a fully synchronous one: A synchronous digital system has the advantage of behaving as a combinational system, piecewise with respect to time. This enables to construct a simple and fast interconnec-

tion network, based on circuit switching that is set on every memory cycle, and not containing any buffers. This interconnection network will be multistage and logarithmic, like the of the baseline or indirect binary n-cube (see e.g. Varma and Raghavendra (1994) [7, Chapter 4]), but duplicated $M_{\text{interleaving}}$ times. The time needed to switch the network should not in itself constitute a sharp constraint on the clock frequency, but to this time there is added the propagation delays of the lines. We now would like to outline a timing estimation. Let T_{prcessor} be the clock cycle length of the processors. To attain a balanced architecture, it is reasonable to fix a memory cycle of $2T_{\text{prcessor}}$. Under a simple organization of the memory system, without pipelining, we will thus have

$$2T_{\text{prcessor}} = T_{\text{switching}} + T_{\text{cache_access}} + T_{\text{propagation}},$$

where the three terms are the times needed to switch the interconnection network, to access the data cache, and to let the signal propagate back and forth on the network lines. The assumption that the interconnection network is logarithmic means that

$$T_{\text{switching}} = \log_2 N_{\text{prcessors}} \cdot T_{\text{single_switch}},$$

where $T_{\text{single_switch}}$ is the switching time of a single switch. We assume that the total length of lines through which a processor is connected to any point on the memory sphere is πR, where R is the memory sphere's radius (at least in the one-dimensional case it is easy to fulfill this assumption—consider Figure 1 again and imagine two mirrored baseline networks). Hence we can write

$$T_{\text{propagation}} = \frac{2\pi R}{\tilde{c}}, \qquad (1)$$

where $\tilde{c}$ is the speed of propagation through the transmission lines or optical lines of the interconnection network. This value is close to the speed of light. Let us observe now that one may write

$$R = \left(\frac{3}{4\pi} \cdot \frac{N_{\text{components}}}{D}\right)^{1/3},$$

where $N_{\text{components}}$ is overall number of components, of various kinds, participating in the macro-architectural level, and D is their mean density in the cased system; the components counted in $N_{\text{prcessors}}$ are the memory modules, interconnection network switches, individual processors, and further components participating in the "super processor" who will be introduced in Paragraph 2.2 and their number is about $N_{\text{prcessors}}$. All in all,

$$N_{\text{components}} = 2N_{\text{prcessors}} + N_{\text{prcessors}} \cdot M_{\text{interleaving}} \cdot (1 + \log_2 N_{\text{prcessors}})$$

where the first term is the number of components in the "super processor" and the second is the number of memory and interconnection network components. The above relations give an estimate for T_{prcessor}, i.e. for the speed of operation of the processors, as a function of the architectural parameters $N_{\text{prcessors}}$ and $M_{\text{interleaving}}$, and of the technological parameters $T_{\text{single_switch}}$, $T_{\text{cache_access}}$, and D. (Note that introducing the mean density D does not

imply that the components are evenly scattered in space). For example, if $T_{\text{single_switch}}$ is 0.4 nanoseconds, $T_{\text{cache_access}}$ is 3 nanoseconds, and the mean density D is one component per 10 cubic centimeters, then we will be able to operate $N_{\text{prcessors}} = 1024$ processors with $M_{\text{interleaving}} = 8$ at a clock cycle length of $T_{\text{prcessor}} \approx 10$ nanoseconds. The radius R would be around 60 centimeters.

We do not claim a scalable architecture, since the quantity $\widetilde{c}$ appearing in Eq. (1) cannot be scaled up. However, an improvement of the technological parameters, and especially of the crucial parameter D, would allow an upgrade of the system.

2.2. The "Super Processor"

The "super processor" comprises

1. individual processors,
2. an apparatus for coordinating the parallel work, called *synchronizer/scheduler*,
3. Global registers for non-synchronization "hot data", with special networks to access them.

The special networks for accessing the global registers serving for non-synchronization "hot data" are spread out in parallel to a *distribution network* that forms a part of the synchronizer/scheduler. Since the synchronizer/scheduler is far more complex than the registers for non-synchronization "hot data", with their associated special networks, we concentrate in this paper only on the synchronizer/scheduler.

The distribution network connects between a *central synchronization/scheduling unit (CSU)* and the individual processors, as shown in Figure 2. Together, the duty of the distribution network and of the CSU is to allocate computational tasks to the processors, while observing the dependency constraints of the parallel program (this is the meaning of synchronization), and while maintaining load balancing and an efficient priority regime. One of the most important parameters of the synchronizer/scheduler is its flow-rate, namely the amount of traffic that can be transferred through cross-section A in the figure. Since the distribution network has a tree topology (with the CSU at its root), it is essential that the traffic via cross sections of ever greater distance from the center will be ever greater. For example, the traffic through cross-section A should be greater than through cross-section B, and the latter should be greater than through cross-section C. Such an amplification effect is attained due to the fact that the traffic in the direction from the CSU to the processors carries chunks of computational work that undergo decomposition as they proceed. These chunks are called *allocation packs.* The decomposition of allocation packs is being performed at each *distribution unit* (a node of the distribution network) using the processor availability state at each sub-tree governed by that distribution unit. Dynamic

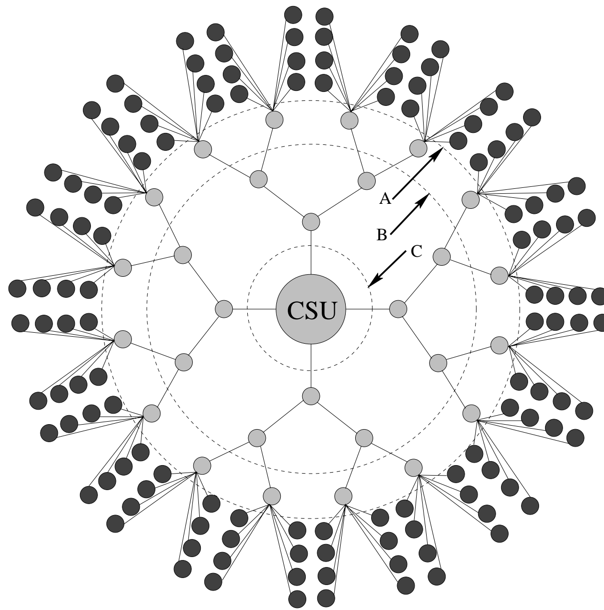

Figure 2: The "super processor". The black circles represent the individual processors, while all the rest is the synchronizer/scheduler. The arrows marked with "A", "B", and "C" point to different cross-sections.

load balancing is thus accomplished. In the opposite direction, namely from the processors to the CSU, *termination packs* are transferred in the network, and are subjected to unification. There exists a similarity between the distribution network and the combining network of the NYU Ultracomputer [3], except that here the unification is not casual but rather dictated by the allocation packs which have passed. Processor availability state updates are also transferred. in the direction from the processors to the CSU, and are also subjected to unification, but in a different way.

In order to describe the structure, mode of operation, and properties of the synchronizer/scheduler in more detail, it is necessary to begin with the system's programming model, on which the very possibility of decomposing and unifying packs of computational work relies. This is the subject of the next paragraph (2.3). Thereafter, in Paragraph 2.4, we briefly describe a possible architecture for the most important unit within the synchronizer/scheduler—the CSU.

2.3. The Programming Model

The program loaded in main memory is essentially a uniprocessor program. It does not contain synchronization primitives. (It may contain references to special registers, however). At the same time, there exists a similarity between this program and a dataflow program: A dependency graph called *task map* that gives a description of a parsing of the pro-

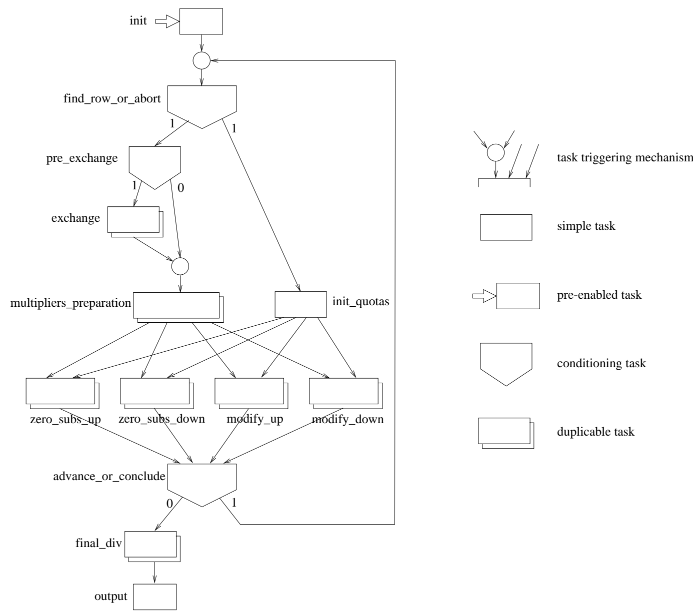

Figure 3: Some graphical symbols serving for describing task maps (at the right), and a complete task map (at the left); this task map belongs to a program that solves linear equation systems.

gram into computational granules, the tasks, and of their interdependencies, forms a part of the representation of the program and is kept by the synchronizer/scheduler. The synchronizer/scheduler uses the task map for scheduling of tasks for execution while maintaining the synchronization mandated from the dependencies. An allocation of a task to a processor is done by transferring the starting address of the task (along with an additional identifier called an "instantiation id" that will be explained later), but the instructions of the task themselves are fetched from memory. A HALT instruction must be implanted at the end of a task. The processor reports on its being in a halted state through a dedicated line, and transmits a *termination condition* bit on another line. The processor may write a value in this bit during run time as if it were a register. When the terminated task is a *conditioning task*, its termination condition serves for managing the global conditioning of the program: Such global conditioning is clearly needed, as the intra-task conditioning using the ordinary branching instructions is not sufficient. Conditioning tasks have a special graphical symbol, that serves when one wants to describe the task map graphically. It is one of several symbols serving for graphical description of task maps, depicted in Figure 3; the figure also depicts a task map. The most important feature of the programming model is *duplicable task*. Such a task is in fact a collection, whose typical average size in an appli-

cation may be quite large, of tasks arranged in a "parallel do" pattern. We refer to these individual tasks as to the *instantiations* of the duplicable task. The instantiation quota of each duplicable task, although physically residing in the synchronizer/scheduler, can be accessed and modified during run time by any processor as if it were a memory word. The instantiations of a duplicable task are implemented by the same instruction in main memory, with the same start address, but the code may contain references to the *instantiation id* kept in a local register. In this way it is possible to create an effect of modifying of the task's code.

This brings us back from the programming model to the implementation: The duplicable tasks are the key for the whole architecture, as the packs decomposed and unified in the distribution network belong to successions of instantiations, derived from the same duplicable task. The task map is kept, in fact, by the CSU.

2.4. The CSU Architecture

The CSU is the most critical component of the whole architecture. There arouse the question of whether at all it can be implemented in a reasonable VLSI technology while attaining reasonable performance. The answer has been provided by developing a prototype CSU architecture possessing the following properties [1]:

- In every clock cycle, the CSU is capable of issuing allocation packs to four branches of the distribution network simultaneously. Every pack contains up to 4K task instantiations (distributed to up to 4K processors known to be available at issue time). The packs are derived from up to four different tasks (which may be duplicable, conditioning, or simple). The generation of the packs is based on full crossing between the tasks ready to execute and the network branches ready to absorb allocation. Likewise, in every clock cycle the CSU is capable of receiving up to four termination packs.
- When a termination pack leads to the the result of issung a new allocation pack, by causing the *enabling* of a new task, the latency between the reception of the termination pack and the issue of the new pack is one to three clock cycles; this holds provided that there is no contention over ports.
- It is possible to load a map of up to 256 tasks in the CSU, of which up to 128 are duplicable tasks, each equivalent to a "parallel do" structure of up to 2^{20} instantiations.
- The CSU can operate at a clock frequency of $f = 1/T$, where T is the propagation delay of a 20-bit carry look ahead adder (built in the same technology as the other circuits of the CSU).
- The implementation requires only about a

quarter of a million transistors. The area of the chip is about 85 square milimeters, in a very conservative 3-micron CMOS technology.

The heart of the architecture is the connection matrix (see Figure 4), in which the task map is coded through programable connections between columns and rows. Each column is mapped to a task. The termination of the task leads to an exitation of the corresponding column. This excitation may propagate along rows and affect *enabling cells* (see the "e-cells" in the figure, at the right of the matrix), which are also mapped to tasks. An e-cell turns on only when all the conditions necessary for enabling the task for execution are satisfied. If the task is duplicable, the cell turns off at the next clock cycle but initializes certain fields in the record belonging to the task within the duplicable task record file. This record file is basically a special, multiported, RAM. The enabling cells and records in an on state feed, via fast multi-output priority encoders, an allocation pack issue unit. That unit is pipelined, and is assisted by a unit for monitoring processors availability state.

3. Summary

Despite the enormous volume of ongoing research, there are still problems that hinder the turning of parallel computation into a practical general purpose technology comparable to serial computing. Overcoming these problems have been identified by David J. Kuck, one of the pioneers of the field of high performance computing, as a major challenge that information society faces.

Contrary to the prevailing trend of distributing the data across the processors and relying on locality, we propose an architectural/physical multiprocessor model based on tight coupling, in the sense that the collection of processors behaves as a single "super processor", working under fine or medium granularity vis-a-vis a symmetric memory. The corresponding programming model is similar to that of a uniprocessor, but also, to a certain extent, to that of a dataflow machine.

Apart
of the processors themselves, the main hardware apparatus contained in the "super processor" is the synchronizer/scheduler, who is responsible for coordinating the parallel work. The key for the efficient implementation of this apparatus and for preventing it from being a bottleneck is the employment of a decomposing/unifying distribution network, which creates an effect of flow-rate amplification. This relies on the presence of "parallel do" patterns within the dependency web of the program. It is possible to implement the building blocks of the synchronizer/scheduler, and the its central synchronization/scheduling unit (CSU) in particular, as "hot" chips: Dedicated, highly optimized, and based on internal parallelism and on pipelining. It is possible to build the CSU around a connectionist structure, slightly reminiscent of a neural network. The synchro-

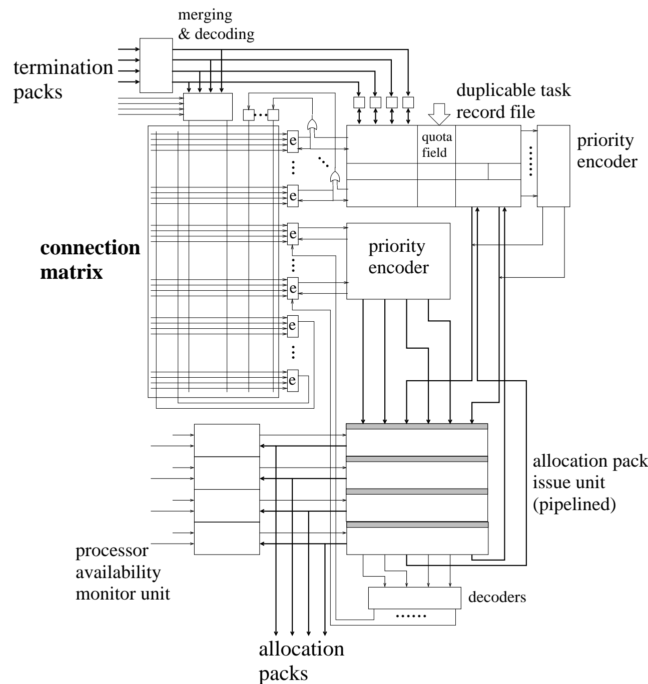

Figure 4: The CSU architecture.

nizer/scheduleras a whole, with its systolic-like structure, also embodies similar traits despite containing a central unit.

The proposed architectural solution for the original problem points to secondary challenges, of a more technological nature, associated with the density of packaging circuitry at the scale of a large system. These may serve as long term major challenges in their own right, much like the miniaturization of single chip circuits has served as a major technological challenge in the 20th century.

References

[1] Peleg Avieli and Oded Rubenov. *Engineering Project Report.* Ben-Gurion University of the Negev. 200 pages, in Hebrew.

[2] David E. Culler and Jaswinder Pal Singh (with Anoop Gupta). *Parallel Computer Architecture.* Morgan Kaufmann, 1999.

[3] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAaliffe, and M. Snir. The NYU ultracomputer—designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, pages 175–189, February 1983.

[4] David J. Kuck. *High Performance Computing.* Oxford University Press, 1996.

[5] Jelica Protić, Milo Tomašević, and Veljko Milutinović. *Distributed Shared Memory—Concepts and Systems.* IEEE Computer Society, 1998.

[6] Top 500 web site. http://www.top500.org.

[7] Anujan Varma and C. S. Raghavendra. *Interconnection Networks for Multiprocessors and Multicomputers—Theory and Practice*. IEEE Computer Society Press, 1994.