Microprocessors: From Desktops to Supercomputers

Forest Baskett and John L. Hennessy

Continuing improvements in integrated circuit technology and computer architecture have driven microprocessors to performance levels that rival those of supercomputers—at a fraction of the price. The use of sophisticated memory hierarchies enables microprocessor-based machines to have very large memories built from commodity dynamic random access memory while retaining the high bandwidth and low access time needed in a high-performance machine. Parallel processors composed of these high-performance microprocessors are becoming the supercomputing technology of choice for scientific and engineering applications. The challenges for these new supercomputers have been in developing multiprocessor architectures that are easy to program and that deliver high performance without extraordinary programming efforts by users. Recent progress in multiprocessor architecture has led to ways to meet these challenges.

The past 10 years have seen phenomenal growth in the performance of low-cost computers that are based on microprocessor technology. Continuing progress in integrated circuit technology and computer architecture will support further rapid improvement in the performance of microprocessors. Just as they have become the dominant force in low-cost computing, microprocessors are becoming the major building block for scientific supercomputers. Future supercomputer-class machines will be parallel computers that use tens to hundreds of high-performance microprocessors.

There are three major challenges to the creation of high-performance, cost-effective computers. First, the rapid growth in microprocessor performance must be sustained. Second, parallel computer architectures that are easy to program and provide scalable cost performance (that is, system performance can be increased by the purchase of more processors) must be developed. Third, algorithms and software that can effectively use these machines must be created.

Here, we begin with a brief review of the trends in integrated circuit technology, the fundamental building block for microprocessors; this review focuses on the technology trends rather than their underlying mechanisms, which are beyond the scope of this paper. Then we turn to the architecture of current microprocessors and advances in both computational and memory systems. Alternative approaches for building multiprocessor systems with high-performance microprocessors, the implications of important scientific methods on the architecture, and some of the programming issues for these parallel machines are then explored, and we conclude with the prospects for continuing growth in computer performance as well as the impact of this growth on users.

F. Baskett is senior vice president of research and development at Silicon Graphics Computer Systems, Inc., Mountain View, CA 94039. J. L. Hennessy is professor of electrical engineering and computer science at Stanford University, Stanford, CA 94305, and chief architect at Silicon Graphics Computer Systems, Inc.

Trends in Integrated Circuit Technology

For the last 30 years, improvements in integrated circuit technology increased the performance and capability of computer systems. During this time, continuous reductions in the sizes of both transistors and wires have allowed an increase in the number of devices that can be put on a single silicon die. Since 1982, the minimum feature size has decreased from 2 μm to between 0.6 and 0.8 μm. This means that the same size die can now contain roughly 10 times as many transistors as before. Furthermore, improvements in manufacturing technology have allowed the die sizes to increase because the number of flaws is now much lower; the maximum economical die size has increased from roughly 0.5 to 1.5 cm on a side. This increase has yielded an additional growth of about a factor of 8 in the number of devices per die. For example, a typical microprocessor in 1982 contained about 35,000 devices, whereas recently announced microprocessors contain approximately 3.5 million (Fig. 1). Decreases in the size of devices and lengths of wires also lead to faster transistors and faster interconnections, which thus lead to improvements in the speed of the chip. Although a linear decrease in feature size can yield a nearly linear increase in speed, a linear improvement in feature size yields a quadratic reduction in transistor area and hence a quadratic increase in the number of devices that fit on a single chip. Thus, the chip architect is motivated to find ways to improve performance that take advantage of this increase in transistor count as well as the faster transistors.

The impact of these improvements in density is most easily seen in memory technology. Such technology has two main types: dynamic random access memory (DRAM) and static random access memory (SRAM). DRAM uses fewer transistors per bit of memory (one transistor for DRAM versus four to six for SRAM) and thus has a higher density. Improvements in memory technology have led to a factor of 4 improvement in density every 3 years, which has in turn led to tremendous increases in the number of bits per chip (Fig. 2). Because of their higher density, DRAMs are cheaper per bit than SRAMs. In fact, because DRAMs have become the primary technology for building main memories, they have the additional benefit of high volume. Thus, they cost typically six to ten times less per bit than SRAMs. This means that for large main memories on computers ranging from personal computers (PCs) to supercomputers, DRAM is the technology

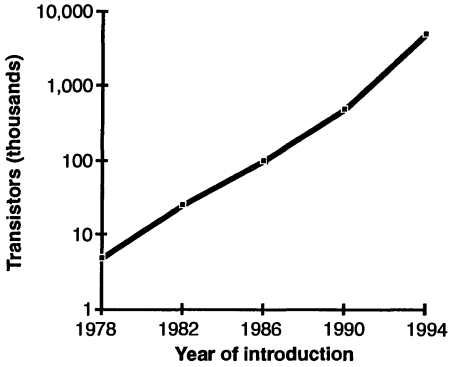

Fig. 1. Growth in the amount of transistors per microprocessor over time for newly introduced microprocessors.

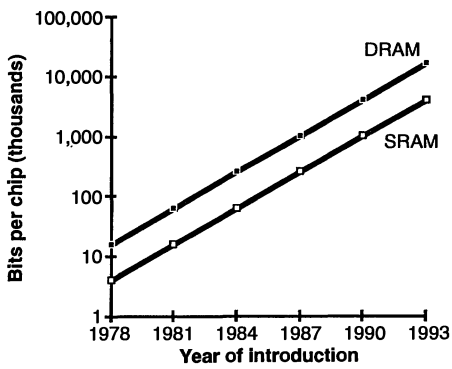

Fig. 2. Growth in the number of bits on a single memory chip for both DRAM and SRAM technologies. Since the first DRAMs were introduced in the early 1970s, memory densities have increased by a factor of 4 every 3 years.

of choice for building main memory. Prices per bit from DRAM have decreased at roughly the same rate as density has increased because the asymptotic cost per memory chip has remained roughly constant. Currently, the commodity price for DRAMs is about $25 per megabyte, which means that each 1-megabit chip is selling for about $3. By 1995, this price is expected to drop to less than $10 per megabyte.

However, DRAM does come with a disadvantage: it is much slower than SRAM. This speed difference arises both from internal differences between SRAM and DRAM and from the standard, low-cost package that DRAMs use; to reduce cost, this package uses fewer pins, which thus leads to slower access time. But low cost and high density, not speed of access, have been the focus of DRAM technology development. For example, since 1980 DRAM access times have decreased by somewhat over a factor of 3, whereas DRAM density has increased by a factor of over 250. Furthermore, microprocessor clock cycle times, which in 1980 were similar to DRAM access times, have improved by over a factor of 100. The situation is somewhat better for SRAM technology, where speed of access is emphasized in addition to low cost and high density. The large and growing gap between the rate at which the basic memory technology (DRAM) can perform an access and the rate at which modern processors can request accesses has led to major challenges in building faster machines. However, as we will discuss later, computer designers have used the concept of memory hierarchies to overcome this memory access limitation.

High-Performance Microprocessors

In designing high-performance microprocessors, designers take advantage of the improvements in technology both to increase the clock rate at which the processor operates and to increase the amount of work done per clock cycle, which is typically measured by the cycles per instruction (CPI). The instruction execution rate is equal to the ratio of the clock rate to the CPI and determines the performance of an architecture (though of course the number of instructions required by different architectures also affects the relative performance of those architectures). For a given architecture, we can maximize performance by increasing the clock rate and decreasing the CPI. Of course, the clock rate and the CPI often trade off against one another: By doing more work in a clock cycle we can lower the CPI (because the number of clock cycles required will be less), but the clock rate may decrease by an equal factor, yielding no performance benefits. Thus, the challenge is to increase clock rate without increasing CPI and to decrease the CPI without decreasing the clock rate.

Clock rate increases have been achieved both through the use of better technology that offers shorter switching delays and shorter connections between switches as well as through better computer architectures that reduce the number of switching elements that must be traversed in a clock cycle (Fig. 3). Over the last 10 years, for instance, the clock rate of microprocessors has increased by a factor of 50; a combination of improved device speed and reduced wiring delays accounts for about 40% of this improvement, and improved architecture accounts for the rest. This improvement by a factor of 50 in microprocessor clock rates over the last 10 years compares to a factor of 3 improvement in traditional supercomputer clock rates in the same time frame. The result is that current microprocessor clock rates are nearly equal to those of supercomputers (Fig. 3); assuming current rates of improvement, microprocessor clock rates should exceed supercomputer clock rates in 1994 or 1995.

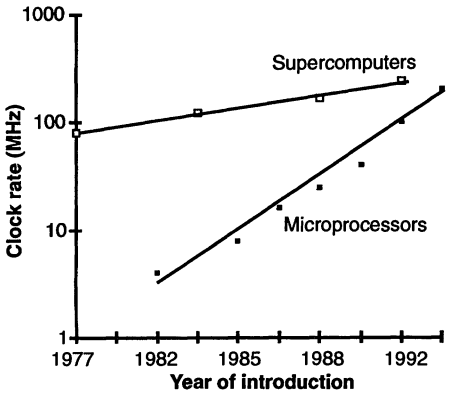

Fig. 3. Improvement in clock rate over time for Cray supercomputers and high-performance multiprocessors.

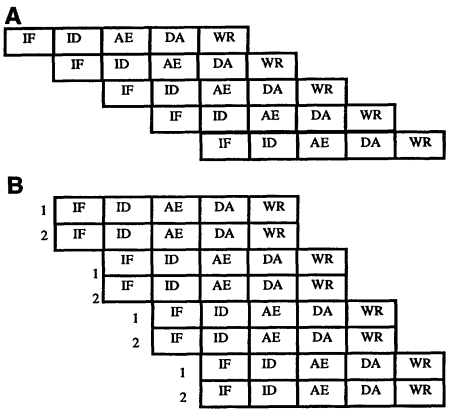

Fig. 4. A typical five-stage pipeline (**A**) and a five-stage pipeline issuing two instructions (1 and 2) every clock cycle (**B**). In (A), every instruction passes through five stages: (i) IF (instruction fetch), retrieving the instruction from memory; (ii) ID (instruction decode), determining what the instruction is; (iii) AE (arithmetic execution), performing an arithmetic operation (that is, for an add instruction or to compute a data address); (iv) DA (data access), accessing memory to retrieve a data item; and (v) WR (write result), writing the instruction result somewhere (for example, into a register). In (B), a multiple-issue machine is depicted with two pipelines that can execute two instructions every clock cycle.

This rapid improvement in microprocessor performance as a result of architectural enhancements really started in the early 1980s. Before that, much of the emphasis in microprocessor design was on functionality enhancement, such as increasing word size (for example, from 16 to 32 bits) or adding more instructions. Designers focused on qualitative measures of an architecture. In the early 1980s, performance started to become a major goal and researchers focused on quantitative metrics such as instruction execution rate and instruction counts. This led to the rapid incorporation of sophisticated architectural techniques that previously were used primarily in large mainframes and supercomputers. The introduction of microprocessors with simpler instruction sets (called reduced instruction set computers or RISCs) allowed these architectural techniques to be incorporated more easily.

Architectural Techniques in State-of-the-Art Microprocessors

To increase clock rate and to decrease CPI, designers can make use of reduced feature size to build smaller and faster processors. Simultaneously, designers can take advantage of the even larger increase in device count to build machines that accomplish more per clock cycle. For scientific computation, a good way to measure how much work is getting done in a clock cycle is to measure the number of floating point operations (FLOPs) per clock cycle. As the number of FLOPs per clock cycle increases, the CPI will fall and performance will increase.

Designers increase the number of FLOPs per clock cycle by implementing architectural techniques that take advantage of parallelism among the instructions. There are two basic ways in which this can be done:

1) *Pipelining.* Here, instructions are overlapped in execution and a new operation is started every clock cycle, even though it takes several clock cycles for one operation to complete. A typical five-stage pipeline is shown in Fig. 4A; such an organization has a throughput that is up to five times greater than that of a machine without such a pipeline.

2) *Multiple issue.* Here, several instructions or operations are started on the same clock cycle, and the instructions can be executed in parallel. A typical pipeline that can issue two instructions in parallel is shown in Fig. 4B. If the instruction pair is restricted to be one integer instruction and one floating point instruction, a multiple-issue pipeline can be built with the addition of only a small amount of logic. Processors that implement multiple issue automatically, without changing the instruction set, are called superscalar processors.

In either case, the overlapping instructions must be independent of one another—otherwise, they cannot be executed correctly at the same time. In most machines, this independence property is checked by the hardware, though many compilers help out by trying to create independent instruction sequences that can be overlapped by the hardware.

Many machines combine these two techniques. For example, vector machines allow a single vector instruction to specify a number of FLOPs that are executed in a "pipelined" fashion. When vector instructions are executed in parallel, multiple FLOPs can be completed in a single clock cycle. Microprocessors have used pipelining for about 10 years, though the earlier machines used little pipelining in the floating point units because transistor counts did not allow it. Pipelined floating point units became the norm in microprocessors about 5 years ago. In the last few years, several superscalar microprocessors have appeared. The result of this evolution has been a steady increase in the number of FLOPs per clock cycle.

Such improvements in integrated circuit technology have enabled microprocessors to perform almost as well as conventional supercomputers. To illustrate this in several different ways, we examined the increase in floating point instruction throughput over time. Figure 5 shows the improvement in the peak number of FLOPs per second that Cray supercomputers and high-performance microprocessors have achieved. By comparison, improvement in the number of FLOPs per clock cycle can be shown with the use of a computational kernel for measurement (Fig. 6).

Additionally, improvement in device speed and integration allows the entire central processing unit (CPU) core to be placed on a single chip, which makes interconnections faster and allows microprocessor clock rates to approach, and in the near future surpass, the clock rates of high-end vector supercomputers. Furthermore, rapid improvements in density have allowed microprocessor architects to incorporate many of the techniques used in large machines for increasing throughput per clock cycle, such as multiple functional units.

Another important advance in microprocessor technology has been the development of microprocessors with 64-bit address spaces. With a 64-bit architecture, the processor can easily accommodate the data requirements of large-scale scientific applications. In fact, this architectural capability means that microprocessors with 64-bit addressing can access more memory than any existing mainframes or supercomputers.

Of course, microprocessors have an enormous price advantage over conventional supercomputers. Because of this and their ability to access large memories, microprocessors are becoming the computing engines of choice for all levels of computing. Later, we will show how this technology can be used to build scalable multiprocessors that offer performance equal to and better than conventional vector supercomputers for less money.

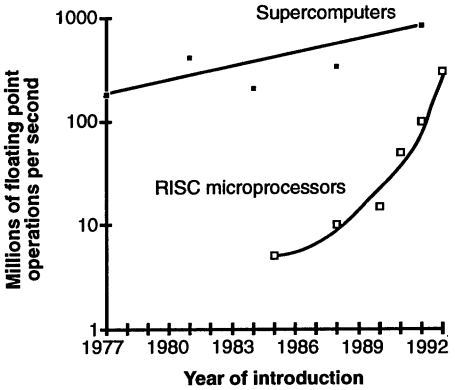

Fig. 5. Peak floating point execution rates (FLOPs) for Cray supercomputers and RISC microprocessors. These rates are the maximum floating point rates and, in general, are not attainable by any real program. Nonetheless, this measurement provides insight into the growth of floating point performance, because application performance tends to track these peak rates.

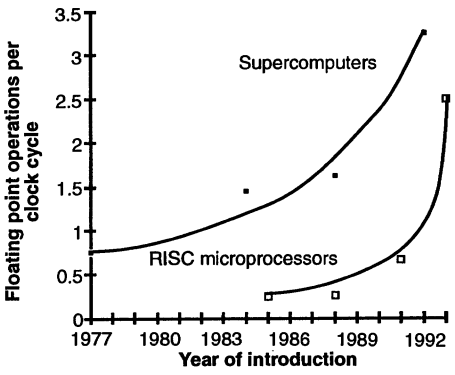

Fig. 6. Floating point operations (FLOPs) executed per clock cycle, with 1000 × 1000 Linpack as the benchmark (*5*). This graph shows the floating point throughput improvement independent of any clock rate improvements.

Meeting the Demands for Memory

To maintain a high instruction throughput rate, a system must be able to satisfy memory requests at a high rate; that is, it must provide sufficient memory bandwidth. Additionally, the memory system should have a low access latency (that is, a small delay per memory access). If there is not sufficient memory bandwidth, the processor will stall because it cannot get the data or instructions it needs. If the average access time (or latency) is high, then the processor will stall because it will run out of things to do while waiting for a memory access to be completed. Thus, the ideal memory system for a high-performance processor would provide 100% of the instruction and data bandwidth required by the processor at as low an access latency as possible. Of course, the memory should also be extremely large. Modern computers achieve this set of seemingly impossible goals by using a memory hierarchy that takes advantage of the typical access patterns of programs.

In scientific and engineering applications, the access patterns for instructions and data are fairly specialized, which allows optimization of the memory system accordingly. For example, for many scientific applications the program is much smaller than the data. Furthermore, the program exhibits high locality of reference (or locality, for short)—that is, only a small portion of the program is heavily used during any given interval. This locality, called temporal locality, arises because the program often consists of nested loops that execute the same instructions many times. Temporal locality can be exploited by trying to keep recently accessed instructions in a place where they can be fetched quickly.

There is often temporal locality in the data accesses, even with very large data sets. An additional form of locality seen in instruction and data accesses is spatial locality. Spatial locality refers to the tendency to make use of instruction or data elements that are close together in memory at the same time. For example, many programs access all the elements in a row or column of a matrix sequentially or consider points in a mesh that are accessed in some sequential order. Spatial locality can be exploited by retrieving memory words that are close to a word that is requested in parallel with the requested word, with the hope that the processor will need the nearby words soon. To take advantage of spatial locality, accesses to nearby data items must also be close together in time. These locality properties do not come from any inherent property of computation but are based on extensive observations of how programs behave. A good way to visualize locality is to think of a plot of the memory addresses

accessed by a program versus time. The presence of clustering in such a plot represents locality; the challenge for the computer architect is to take advantage of this clustering.

Given the presence of locality in memory accesses and the trade-off between SRAMs and DRAMs of speed versus cost, how can a memory system be organized so that it meets the demand of a high-performance CPU and also takes advantage of low-cost DRAM technology? The answer lies in using a hierarchy of memories with the memories closest to the CPU being composed of smaller, faster, but more expensive memory technology. A memory hierarchy is managed so that the most recently used data are kept in the memories closer to the CPU; these memories, which hold copies of data in the levels further away from the CPU, are called caches. The lowest level of the memory hierarchy is built with the lowest cost (and lowest performance) memory technology, namely DRAMs. A typical structure in a memory hierarchy and some typical sizes and access times are shown in Fig. 7. The registers and often the first-level cache are today contained on the microprocessor. The goal of this organization is to allow most memory accesses to be satisfied from the fastest memory, while still allowing most of the memory to be built from the lowest cost technology. Temporal locality (clustering on the time axis) is exploited in such a structure, because newly accessed data items are kept in the top levels of the hierarchy. Spatial locality (clustering on both the time and address axes) is exploited by moving blocks consisting of multiple words with adjacent addresses from a lower level to an upper level when a request cannot be satisfied in the upper level.

A memory hierarchy achieves both high bandwidth and low latency of access. The former is accomplished by satisfying most requests from the fastest memory that can support a high access rate. Similarly, the top level of the hierarchy also has the lowest access time, which thus leads to low overall latency. Today, machines ranging from low-end PCs to high-speed multiprocessors use caches as the most cost-effective method to meet the memory demands of fast CPUs.

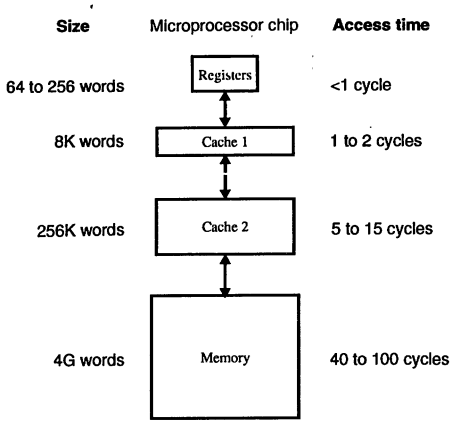

Fig. 7. A typical memory hierarchy, showing the increase in size and in access time through levels of the hierarchy. Although not shown here, bandwidth is also usually higher in the upper levels of the hierarchy.

Vector supercomputers also use a memory hierarchy, but most such machines do not contain caches. Instead, most vector supercomputers contain a small set of high-bandwidth, low-latency vector registers and provide explicit instructions to move data from the main memory to the vector registers in a high-bandwidth bulk transfer. The goal in such a design is to be able to move an arbitrary vector from memory to the CPU as fast as possible. Moving an entire vector takes advantage of spatial locality. Once the vector is loaded into the CPU, the compiler will try to keep it in a vector register to take advantage of temporal locality. Because there are not many vector registers (8 to 64 typically), only a small number of vectors can be kept close to the CPU.

There are important differences between these two approaches. Because the vector registers are relatively small, many data accesses in a vector machine need to go to the main memory. A memory hierarchy with caches can provide a much lower average latency because most accesses will go to the top level of the hierarchy, which has a much shorter access time than the main memory. A major benefit of the shorter access time is that the processor can be kept busier because it does not have to wait for the memory. To minimize the long latency of memory access in a vector supercomputer, many such machines construct their main memory from SRAM. This reduces the time to access this large memory and makes it easier to provide high bandwidth. This approach, however, is much more expensive, as SRAM is 6 to 10 times more expensive per bit than DRAM. Thus, memory systems in vector supercomputers are often as much as 10 times more expensive per bit than the memory systems of microprocessor-based machines with cache-based memory hierarchies. Another important advantage of a memory hierarchy is that its hardware manages the caches automatically, whereas use of vector registers requires assistance from the compiler or the programmer.

The major disadvantage of a cache memory hierarchy is that it typically provides lower bandwidth to the main memory (the lowest level of the hierarchy). This means that programs that do not exhibit good locality will be penalized. Although this has been a major drawback, recent progress in algorithms and compiler technology has led to the development of methods for improving the locality of access for arrays, which are the primary data structure in scientific programs. Most important scientific algorithms can be adapted in this way, although automatically applying these techniques within a compiler has been accomplished to date only for a narrower set of problems. One technique, called blocking, is able to significantly reduce the number of requests to the main memory, making caches work extremely well. The idea behind blocking is to restructure the computation so that the memory accesses are clustered in both the time and address dimensions. For example, matrix multiply is blocked by transforming a straightforward version of the operation into a version that operates on submatrices, which thus computes partial results. Another technique, called prefetching, tries to reduce or eliminate the penalties encountered when the data items accessed by a program will not fit in the cache. With prefetching, the compiler determines that certain data items not in the cache will be needed in the future and signals the memory hierarchy to fetch these items into the cache before they are actually needed. The fact that caches are typically larger and have a more general purpose than vector registers makes cache-based memory hierarchies more efficient over a larger range of computing problems that the vector register–based systems, even though the vector machines often have a faster access to the main memory.

Parallel Processing with Microprocessors

As microprocessors become the dominant type of processor for science and engineering computations, with system costs typically more than 10 times lower than the costs of conventional supercomputers, it is natural to ask if multiple microprocessors can be used in parallel for single problems to further increase speed and to make possible new and more ambitious applications. Research over the last 10 years has demonstrated this potential in most of the application areas of science and engineering.

There are four major computational methods in use within the range of disciplines that study the physical world from the smallest atomic distances to the largest galactic distances. The multipole methods have recently revolutionized computational dynamics at the atomic, molecular, and galactic levels (*1*), though these techniques are also applicable at other scales. The three other classes of solution techniques are direct matrix methods, iterative meth-

ods on discrete grids, and spectral methods.

For each of these four computational methods, parallel versions have been successfully developed and are being used. Parallel versions of spectral methods such as the fast Fourier transform have been in use for many years. Multipole methods, which treat clusters of particles and summarize the effects of one cluster on the other with the moments of the cluster, are naturally parallelized because each cluster is treated in parallel. Iterative methods on discrete grids can be parallelized by dividing the grids into subgrids that are processed in parallel, and neighboring subgrids can communicate the new values of boundary points after each iteration. Parallel versions of multigrid and adaptive multigrid techniques are also being developed. Obtaining good performance on direct methods was extremely challenging, not because of the lack of parallelism but because of the high cost of data communication. However, the development of blocking (or tiling) techniques has overcome the communication problem, leading to highly parallel versions of many common linear algebra methods.

Because parallelism seems readily available, what must be addressed next is what type of parallel processors will prove most useful for these applications. From a distance, all existing large-scale parallel machines look essentially the same: they consist of processor-memory pairs connected together by an interconnection network, which is used for interprocessor communication. Such a machine is called a distributed memory machine. There also exist machines with a single centralized memory, which typically use a single bus to connect all the processors and the memory. These bus-based machines have been extremely successful with up to a few tens of processors, although the use of a single centralized memory and a bus interconnect does not allow such machines to have larger numbers of processors. Nonetheless, this bus-based organization will be the architecture of choice for small processor counts for several more years.

Two features that are crucial in determining the effectiveness and ease of use of a parallel processor are the method used for interprocessor communication and the organization and performance of the interconnection network. The communication mechanism and network are critical because communication is far more expensive than computation. In current machines, communication delays can be from 100 to 10,000 processor clock cycles. Although improvements in the absolute communication time are likely (especially at the high end of the scale), ongoing processor enhancements are likely to mean that communication will continue to cost at least 100 processor clock cycles on high-performance machines. To understand how to optimize this expensive communication in both the application and the architecture, it is necessary to understand the communication characteristics of the major computational methods.

Requirements of Parallel Applications

For a parallel machine with P processor-memory pairs, the parallel characteristics of each of the four computational methods are summarized in Table 1. For problems of size N (where N is the number of unknowns), direct methods have substantial communications requirements because every submatrix must be communicated to a subset of the processors, but the computational requirements dwarf the communication requirements for large problems. The spectral methods have the highest ratio of communication to computation and thus present the most difficulties for efficient implementation. (Spectral methods also use complex and costly communication patterns, which poses additional challenges to implementation.)

Our focus in this paper is on the application of parallel processing to single, large-scale problems (where N is large) that tax the computational power of the fastest single-processor machines currently available (that is, such large-scale problems might run for many hours or even days on such machines). To obtain significant performance advantages for such problems, a reasonable number of processors needs to be used. With large numbers of processors, the communication-to-computation ratio (Table 1) will be an important factor when the higher cost of communication is accounted for. Thus, machines with high-latency, low-bandwidth communication mechanisms, such as a local area network interconnection, will not perform very well in such applications.

Communication is also characterized by the frequency and amount of data communicated among parallel processes. In some methods, data is communicated less frequently but in large quantities; methods with this characteristic are called coarse-grained. Other methods communicate in a less structured fashion with smaller amounts of data communicated more often; this type of communication is called fine-grained. Of course, the distinction between fine- and coarse-grained computation is not rigid, and different implementations of a method may have different granularity. Nonetheless, the distinction is important in evaluating the communication mechanism. To the extent that a program favors more fine-grained communication, a communication structure that achieves high bandwidth only when communicating large blocks of data will not be efficient. Small-scale multiprocessors have been effective at handling fine-grained communication, because the processors can be closely coupled and the communication has low latency (that is, a small delay until completion). Large-scale machines have been better matched to coarse-grained parallelism because the overhead of initiating communication has been much higher (by factors of 100 to 1000) than on small-scale machines.

One potentially attractive approach to building parallel processors is to use workstations connected on a local area network, often called a workstation cluster. This approach, however, has proved suitable only for applications where the parallel computations are so coarse-grained as to be essentially independent. A classic example of such a computation is a problem that involves many independent simulations. For this type of application, the workstation cluster is extremely cost effective because a

Table 1. A comparison, for four major solution techniques, of the scaling of computation required, parallelism available, necessary communication, and the ratio between communication and computation. P is the number of processors; N is a measure of the size of the problem. For direct methods, N is the number of unknowns; for iterative methods, N represents the size of one side of the grid. For both multipole and spectral methods, N is the number of sample points.

Method	Problem size	Computation needed	Parallelism available	Communication required	Ratio of communication to computation
Direct	N^2	N^3	N^2	$N^2 \times \sqrt{P}$	$\frac{\sqrt{P}}{N}$
Iterative	N^2	N^2	N^2	$N \times \sqrt{P}$	$\frac{\sqrt{P}}{N}$
Multipole	N	N to $N \times \log N$	N	$\sqrt{N \times P}$	$\frac{\sqrt{N \times P}}{N}$
Spectral	N	$N \times \log N$	N	$N \times \log P$	$\frac{\log P}{\log N}$

high-speed communications network is not required. If one tried to use a workstation cluster for the type of problems shown in Table 1, the performance would be very poor except for problems where the ratio of problem size to number of workstations was very large. Although a cluster might operate efficiently in such a situation, the small number of processors compared to the large computational requirement would mean that the time to complete the run would be very long. Currently, both the lower bandwidth of typical local area networks and their high communication latency limit workstation clusters to a narrow class of parallel applications. This class can be expanded in some cases with methods that are more coarse-grained, although this often requires substantial programmer effort. Effectively executing large, single parallel applications with the methods listed in Table 1 requires higher bandwidth, lower latency interconnection technology.

An important side benefit has emerged from the development of parallel methods. Newly developed parallel methods have naturally emphasized an arrangement where one processor would work on one submatrix, subgrid, cluster of data elements, or subarray. This arrangement (sometimes called locality of computation) is necessary (and natural) to limit the amount of communication required to support the parallel computation. This locality of computation also results in better locality of reference to the data. Thus, these parallel methods with improved locality often use the memory hierarchy more efficiently. In fact, the improved locality of the parallel version often leads to shorter execution time when that version is run on a single processor because of the improved performance of the memory hierarchy. This speedup also represents an important lesson for those who have developed computational methods on vector-style supercomputers. In many of those methods, locality of reference is at odds with the efficient use of vector registers and the high-bandwidth memory systems to which they are connected. As a result, vector supercomputer applications do not necessarily port efficiently to parallel microprocessors without restructuring data accesses to improve locality of reference.

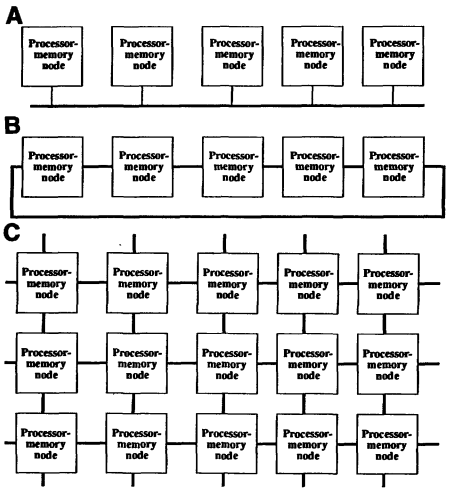

Fig. 8. Three typical interconnection networks. (**A**) Bus: a zero-dimensional interconnect. (**B**) Ring: a one-dimensional interconnect. (**C**) Mesh: a two-dimensional interconnect.

Interconnection Technologies for Parallel Processors

Interconnection technology for parallel processors has taken a wide variety of forms. These different forms emphasize trade-offs in cost, bandwidth scalability, and efficiency. For example, in smaller scale multiprocessors, cost and bandwidth per processor tend to be crucial. In larger scale machines, total system bandwidth becomes crucial, and a designer may favor scalability of communications bandwidth over local, per processor bandwidth and cost. Interconnection schemes that scale well tend to sacrifice per processor bandwidth and are most costly, whereas schemes that are inexpensive often do not scale. Looking at the range of current machines and the fundamental properties of interconnection networks, it appears quite difficult to design interconnection schemes that satisfy all the desired goals.

Another important trade-off is bandwidth versus latency of communication for small data items. Lower communications latency means that a machine can take advantage of finer grained parallelism. In addition, lower communication latency usually eases the job of the programmer, because less effort is required to hide or overlap the communication delays. Recent trends in interconnection technology have tended to favor smaller communication units (called packets), which has led, as we shall discuss later, to a situation where the method used to communicate (rather than the interconnection technology) is often the major source of delay.

Table 2. Interconnection networks of different dimensions connecting P processor nodes. The bandwidth measures show how bandwidth scales with processor count (P); the last column shows how the number of wires (a good measure of cost) scales with the number of processors. C, constant.

Interconnection network	Dimensions	Maximum system bandwidth	Bisection bandwidth	Wire count per data bit
Bus	0	C	C	1
Ring	1	P	C	P
2D mesh	2	P	$\sqrt{P}$	$4P$
3D mesh	3	P	$\sqrt[2/3]{P}$	$6P$

Possibly the most important trade-off in designing an interconnection network involves bandwidth and cost. Bandwidth can be measured with two different metrics: how the bandwidth scales in the best case when communication is localized and how it scales when the network is used for a random communication pattern. The former is measured by computing the maximum aggregate bandwidth under ideal conditions (typically, nearest neighbor communication). The latter is typically measured by computing the bisection bandwidth, which is the bandwidth available across a bisection of the processors into two equal parts. One simple way to assess cost is to count the number of wires, because wires are one of the most expensive parts of an interconnection network and their cost is proportional to that of other costly parts, such as network ports.

Figure 8 shows three of the most common interconnection networks in use today. The dimension of a network refers to the number of dimensions through which data can flow at once. For example, in a bus communication is broadcast on the bus and all data flows through a single point. In contrast, meshes can be expanded into more dimensions; for example, the upcoming Cray T3D uses a three-dimensional mesh. With the exception of the bus interconnect, which is not scalable because the bandwidth does not increase as the number of processors increases, all of the interconnection networks in Fig. 8 are indirect networks. This means that each node in the network contains a processor, and data is routed through the nodes to reach other processors. There are also direct networks where the intermediate nodes between the source and destination are all switches (such as small crossbars) and are not processing nodes also. Such direct networks are more complex and more costly than indirect networks but have the advantage that bandwidth, especially bisection bandwidth, tends to scale better. Table 2 shows how communication bandwidth and the number of wires scale with the number of processors in some of these indirect networks. It is clear that different networks may be more appropriate for different system sizes, because the trade-offs between cost and scalability are substantial.

The Communication Method

Conceptually, microprocessor-based multiprocessors seem to be in great shape: The technology for building these machines is

progressing rapidly as is the understanding of parallel solution methods. The question of how processors should communicate data is one of the most important remaining questions, because the communication method affects both the programming model and the cost of communication. The importance of the programming model cannot be overstated: Many multiprocessors have gone unused not because they had major flaws in the processors or the interconnection, but because they were simply too difficult to program. The major choices for the communication method are shared memory and message passing.

The primary advantage of message passing is that it is simple and cheap to build. Little or no additional hardware is required beyond the processors, memory, and interconnection network. For this reason, most of the large-scale multiprocessors to date have used this communication model. From the programmer's viewpoint, message passing has the severe drawback of forcing the programmer to partition a program into separate processes that communicate explicitly by sending messages rather than implicitly through memory.

In contrast, the shared memory model (more appropriately called the shared-address space model) allows the programmer to directly reference data in any of the physically distributed memories, independent of the location of the data. The programmer does not have to partition the data and insert messages to get the program to run, although locality of reference is important for good performance. Fortunately, the task of improving locality is also simpler with a shared memory model and appropriate hardware support.

The shared memory model is a natural extension of the uniprocessor programming model and is therefore much more familiar to programmers and more easily supported in standard programming languages. It is also the standard model for small-scale, bus-based multiprocessors. Shared memory communication can also be more efficient than message passing because it can be completely supported in hardware with the use of existing techniques of memory hierarchies.

Researchers and designers realized early on that scalable parallel machines would require memory to be distributed independent of its logical sharing. With that realization, it was quite natural to hook the processors together with a communication mechanism based on the input-output (I/O) support already existing in the processor. Thus, communications were treated like I/O and parallel processes communicated by passing messages through the I/O channel. Recent message-passing systems have greatly streamlined this communications path, but existing systems all have a lot of overhead involved in communications. This overhead makes fine-grained communication very costly and forces the programmer to work hard to avoid communication and to organize any remaining communication into large blocks.

Although shared memory models may be easier to use, the challenge for the shared memory model has been to find a method to scale to large numbers of processors, organized with physically distributed memory. In the last several years, we have learned how to do this. The resulting architectural approach, called distributed shared memory (DSM), can be made to work with almost any communications structure. When a data access is attempted by a processor, the memory system determines whether the access is to a local memory or to a remote memory; if the access is remote, the memory module generates a request to get the data from the remote memory. This request is routed over the interconnection network to the remote memory, where the data is retrieved and then sent back to the requesting processor. Logically, the application has access to all the memory in the machine, and although remote accesses are more expensive than local accesses, this hardware-supported mechanism has a much smaller overhead than message-passing communication.

An idea closely related to DSM is distributed virtual memory (DVM) (*2*). Distributed virtual memory makes use of standard local area networks and virtual memory hardware to create a shared memory among physically separate machines, such as a cluster of workstations. A major advantage of DVM is that it unifies the software model for multiprocessors and workstation clusters. Its major drawback arises from the limitations of the underlying hardware, which typically provides limited bandwidth and has high communication latency. Another difficulty in using DVM is that the unit of communication, namely pages in virtual memory, is mismatched to the communication needs of most applications. In some cases, a page is too large a unit, and in other cases, it is important to optimize the page boundaries, which are typically invisible to programmers, thus introducing highly machine-dependent details into the program.

The amount of remote communication in a DSM machine can be reduced by caching. By simply allowing remote data to be placed in the cache of a processor, one can greatly reduce the access time for subsequent accesses to this data. But the caching of data shared by multiple processors does introduce a new problem: cache coherence. Such a coherence problem arises because a shared data item may be read and written by a number of processors. The system must ensure that the value read by a processor is the most recent value written for that item. In small-scale, bus-based multiprocessors, this problem was solved more than 10 years ago with a technique called snoopy caches. In a bus-based system, every processor can see every memory access. It is therefore simple to have each processor update its copy of a data item when it sees the item is being changed by a transaction on the shared bus. Because caching is an attractive method to reduce latency and because it would be unacceptable to burden the programmer with the task of keeping caches coherent, the development of snoopy caches was key to making bus-based multiprocessors effective. Unfortunately, the mechanism used in snoopy caches is not scalable to large numbers of processors because it relies on communicating with every processor on any update of a shared data item, whether a processor has a copy of the item.

A major challenge in developing DSM architectures has been to deal with the cache coherence problem. DSM multiprocessors without cache coherence were developed early; however, these machines did not prove popular largely because of their inherent programming difficulty, which was similar to that required by message-passing architectures. Only in the last few years have DSM machines been built that can support cache coherency in a scalable fashion with the use of an arbitrary interconnection network. Furthermore, the cost of the additional hardware needed to keep the caches coherent is modest. The Stanford DASH machine (*3*), for example, estimates the added cost to be between 10 and 15%. Although this is a noticeable cost increment, the benefits in improved programmability and achieved efficiency have been enormous.

This improvement in our understanding of how to build scalable parallel processors comes at a very propitious time. Technology trends are making it less and less attractive to have centralized shared memory for systems with even moderate numbers of processors. But with this new understanding, it is now possible to supply a single, shared memory programming model independent of the underlying interconnection technology, the number of processors, and the physical distribution of memory. Thus, architects designing a new multiprocessor system can treat the choice of interconnection technology and topology as an engineering problem and can expect parallel applications to run well with few or no changes.

The shared memory programming model is analogous to the virtual memory programming model. Virtual memory is a good thing as long as it is used appropriately; likewise, shared memory is a good thing as long as you use it intelligently. Locality of

computation and locality of reference continue to be critical for efficient use of these systems. To help the user there are programming environments that contain a variety of tools, some automatic, for developing better parallel applications in an incremental fashion. It is possible to start with a serial version of a method and slowly change it into a good parallel method. The parallel version can then be optimized to improve locality and enhance performance. This approach is in contrast to message-passing systems, where great up-front efforts are required just to implement parallel versions that simply run correctly and incremental progress is rare. In addition, the interconnection structure and communication characteristics in message-passing machines have been so critical to performance that programs had to be developed and tuned with one particular machine in mind. Even porting message-passing programs from one parallel machine to another could require a significant effort. A common story is that of William Goddard, a colleague in chemistry. His team spent 6 months trying to develop a parallel version of a multipole method on a message-passing machine. They gave up and then spent 2 weeks successfully developing an efficient parallel version on a DSM system. Similar experiences have been found for a variety of n body applications (4).

With the success of microprocessor hardware, from the desktop to a new generation of supercomputers, we now also have a single programming model capable of spanning the entire range of microprocessor-based computing systems. Furthermore, methods that emphasize computational locality so as to reduce communication also naturally lead to better locality in data referencing patterns, which better uses the memory hierarchy.

Future Trends and Implications

The rapid progress in microprocessor performance since the early 1980s is likely to continue for at least five more years. Improvements in integrated circuit technology promise not only improved clock rates but also substantial growth in the number of transistors per processor: microprocessors with clock rates of 400 MHz and higher and over 5 million transistors are less than 5 years away. Using these increased transistor counts, architects will employ a variety of techniques to further increase the number of instructions executed every cycle and to decrease losses from the memory hierarchy and from inefficient use of pipelines. Together, the technology and architectural enhancements should drive an annual performance growth rate of at least 50% and possibly closer to 100% (at least for floating point programs) during the next several years.

This ongoing rapid technology improvement will further reinforce the role of the microprocessor as the dominant computing element. Furthermore, it has important design and economic implications for how large-scale parallel processors are constructed. The rapid performance growth of microprocessors leads to rapid obsolescence of machines built from microprocessors. The industry and users have adapted to this situation on the desktop by using 3-year depreciation cycles for this rapidly evolving technology as well as planning upgrades to allow the basic hardware to be used for more than one processor generation. For large computing systems, however, the common practice has been depreciation schedules of 5 years or even 7 years. If our large computing systems are microprocessor based, these systems will become obsolete when the microprocessors inside them become out of date in 3 years. Whereas most large machines allow for expansion and upgrade of memory and peripherals, the use of microprocessor technology motivates designs where the processor (and often the caches) can also be upgraded. In a typical microprocessor-based computing system, the microprocessor and its cache or caches account for approximately 25% of the total cost, whereas the memory system, interconnect, peripherals, cables, cabinets, fans, and power supplies account for the remaining 75%. A design that supports upgrading to a newer processor model may have some additional initial cost, but it can have a significantly longer useful life. Unfortunately, although it is possible to build a system that will allow an upgrade that will double or perhaps triple performance, it is difficult and costly to design a system that will last through more than one upgrade. One major reason for this is that upgrading the interconnection technology is typically very difficult, and designing an interconnection technology to support several upgrades usually becomes too costly.

Beyond 5 years, the prognosis for the performance growth of microprocessors is more hazy. The scope of the architectural techniques being used to extract performance from a single instruction stream has not grown much in the past few years. There also appear to be major engineering limitations in our ability to build processors that exploit unlimited amounts of instruction-level parallelism (the type of parallelism exploited by pipelining and multiple issue). If these limitations are not overcome—a task that today looks quite difficult—performance growth for microprocessors may eventually be limited primarily by technology-driven clock rate enhancements. Such a situation will likely lead to performance growth rates of about 25 to 30% per year. Indeed, this is the growth rate one could see if one looked at high-end mainframes or supercomputers that have exploited many of these architectural enhancements already.

Such a slowdown in the performance growth of uniprocessors will further increase the importance of multiprocessors, especially for scientific and engineering computations where parallelism is generally in abundance. A major challenge for users of parallel computing has been the variety of incompatible programming models and architectures. Because parallel computing is still maturing, the development of a single programming model that can span multiple generations of architectures as well as a range of processor counts is critical. Fortunately, we have learned how to build large-scale multiprocessors that support shared memory, the model of choice for both uniprocessors and small-scale multiprocessors. This advance should lead to a single programming model supported on a wide variety of different architectures.

Although a single programming model greatly improves our ability to use these new parallel machines and to reap the benefits of a software investment, parallel processing still presents challenges to scientists and engineers who would take advantage of it. In particular, it will be critical to develop new parallel methods that exhibit locality so that the machines can be used efficiently. Users who understand the importance of both parallelism and locality will be able to use these cost-effective multiprocessors to perform computations much more cheaply than would have been possible just a few years ago with conventional supercomputers.

REFERENCES AND NOTES

1. L. Greengard and V. Rokhlin, *J. Comput. Phys.* **73**, 325 (1987).
2. K. Li, *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, 15 to 19 August 1988, H. E. Sturgis, Ed. (Pennsylvania State Univ. Press, State College, PA, 1988), vol. II, pp. 94–101.
3. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, *IEEE Comput.* **25**, 3 (1992).
4. J. P. Singh, thesis, Stanford University (1993).
5. J. Dongarra, "Performance of various computers using standard linear equations software" (Technical Report, Computer Science Department, University of Tennessee, Knoxville, TN, 1992).

6. We thank the referee for insightful and helpful comments. We also thank J. Winget and J. P. Singh for their careful reading and helpful comments.