ICAP/3090: Parallel processing for large-scale scientific and engineering problems

by E. Clementi
D. Logan
J. Saarinen

Described is the ICAP/3090 (for loosely coupled array of processors) parallel processing system. General parallel processing performance issues that determine the success of all multiple-instruction/multiple-data-stream parallel computing systems are examined in the context of large-scale scientific and engineering problems. Experiments with previous ICAP parallel processing systems that have made possible the present design of ICAP/3090 are also described.

In the last several years, it has been generally agreed that in the future supercomputing and parallel processing will be to a large extent synonymous. This is evidenced by the fact that systems such as the IBM 3090 multiprocessor family and the CRAY-XMP series are now being increasingly employed in a parallel mode to solve problems that cannot be feasibly solved on a single processor. It is also clear that parallel supercomputers must employ a balanced approach, using nodes with high-performance scalar and vector capabilities. In short, as a concept, supercomputing is the union of all approaches that maximize performance.

However, there is at present little consensus on the optimal architecture of parallel processing systems. Roughly speaking, designers have chosen one of two broad approaches that can be categorized as either shared- or distributed-memory systems. Representative of shared-memory systems are the previously mentioned products, and representative of distributed-memory systems are the large number of commercial hypercube offerings. However, more radical approaches to parallel processing also include systolic and wavefront array processing; very-long-word, multiple-instruction machines; and data-flow systems. Most of these more radical systems have not been marketed commercially, with the exception of the Warp machine[1] (systolic array) and the Multiflow processing system (very-long-word instruction).[2]

Another distinction between types of parallel systems is whether they purport to be general- or special-purpose systems. The general-purpose systems include most of the products previously mentioned. However, many such systems have been built without a clear focus on the types of applications to which they will be put, and thus the mapping of applications to machine has often become an experiment in parallel processing after the fact, with little guarantee of success.

The alternative approach, i.e., the design of special-purpose parallel machines, is targeted to a very limited class of applications. Included here are a number of systems tailored to solve problems in fundamental particle physics (the IBM GF11 machine[3]) or applications in fluid dynamics (the Navier–Stokes machine[4]). The design process has usually been a bottom-up approach, with specific layouts of chips, hard-wired interconnection networks, and special memory devices. Thus, as with the general-purpose systems, the elapsed time between design and inception is large.

An exception to these approaches has been the development of the ICAP parallel processing system. This system was conceived with a clearly defined set of applications in mind. The goal was to solve these problems by using parallel processing at a rate comparable to, but at a cost less than, that of the fastest commercial supercomputers of the day. The class of applications was broad enough not to be considered as special-purpose usage but, on the other hand, could not be classified general-purpose usage either. This area comprised large-scale calculations in theoretical chemistry. The approach taken in the design of this system was to minimize the hardware and software complexity of the task, and thereby minimize the time needed to build the system, by simply interconnecting commercially available offerings. The objective was therefore not to conduct an experiment in parallel processing, but rather to use parallel processing to solve real problems.

Our goal was quickly achieved;[5,6] this was not surprising, because we had realized beforehand the requirements of our applications and their large-grain-size, parallel structures. Thus we considered the design of this system obvious and not worthy of a large treatise on the theory of parallel processing. Indeed, many results have been published from calculations performed on the system, often without mentioning that it was a parallel computer. We were interested in results rather than means.

Because of this success, we began to explore whether we might meet with equal success in areas other than theoretical chemistry. In short, we wanted to see whether our system, which we call ICAP (for loosely coupled array of processors), could be considered a general-purpose, parallel computer. This meant surveying a large spectrum of applications in science, engineering, and mathematics and attempting to adapt them to ICAP. A program for visitors was instituted to solicit professionals, skilled in areas that we were interested in studying, to experiment with parallelism. Toward the same end, our department was expanded to include personnel with backgrounds other than theoretical chemistry. This led to ICAP evolving, in both hardware and software, into a general-purpose, parallel computer.

The outcome of our experiments is culminating in the current development of the ICAP/3090 system. A brief description of this system and parallel language issues is given in the next section. We then discuss general principles of parallel processing that apply to all computers of the Multiple Instruction/Multiple Data stream (MIMD) variety. This classification includes all parallel computers that have multiple processors that can execute different instruction streams or programs, using unique data or otherwise, in the solution of a single problem. Following this discussion, we describe the experiments in parallel processing that have led to our ICAP/3090 efforts. Finally, we offer our conclusions.

The 3090 multiple-processor systems have been quite successful in increasing system throughput.

The ICAP/3090 system

The IBM 3090 vector multiprocessor family encompasses a variety of models ranging from a two-processor system (Model 200) to a six-processor complex (Model 600). These systems can increase the throughput of a large workload by using the multiple processors on independent jobs. This has been the traditional motivation behind the development of such systems. An important corollary of this approach is that memory must be increased proportionately. This contributed to the development of expanded storage on the 3090 to alleviate the effect of paging that occurs when multiple jobs compete for real memory. Overall, the 3090 multiple-processor systems have been quite successful in increasing system throughput.

The performance of the 3090 base system exhibits a marked improvement over its earlier counterpart, the IBM 308X system. Scalar computing capability was enhanced not only by reducing cycle time from approximately 27 to 15 nanoseconds, but by increasing the degree of overlap between instruction fetch, lookahead, decode, and execute. Also, vector capabilities were introduced in an integrated architectural manner with the addition of vector registers and 171 instructions. Overall, the design of this system emphasizes a balance[7,8] of memory access, I/O, and processing capabilities. This has led to a vector capability that seems to be optimally cost-effective in addressing problems of an approximately 50–80 percent vector content. This is representative of the majority of scientific and engineering applications used in industry and universities.

The ICAP approach is to increase still further the level of parallelism.

The performance potential of a 3090 multiprocessor can also be realized by allowing multiple processors to work on a single problem. This solution is limited by the degree of parallelism that exists in scientific problem codes and the extent to which performance becomes degraded when processors need to communicate. These issues are explored in greater depth in the section on parallel processing performance.

To address utilization of the multiple processors of the 3090, a parallel FORTRAN capability, called the Multitasking Facility (MTF), was initially developed for these systems. While useful, it was limited in availability to MVS installations, and limited functionally in that parallel execution was limited to a simple fork/join capability. More recently, a much richer set of parallel capabilities, including automatic-compiler-generated parallel code, has been developed. This set of software, called the Parallel FORTRAN (PF) package, is now proving its worth for a large variety of applications.

The ICAP approach is to increase still further the level of parallelism. We intend to couple readily available commercial processors (in this case, IBM 3090s) to form a system that is not massively parallel, but rather is modular and can be expanded to match the degree of parallelism that a set of applications can support.

ICAP/3090 architecture. The principal idea of ICAP/3090 is to couple together clusters of IBM 3090 multiprocessors. We intend this coupling to be extendable, so that, as faster versions of IBM multiple-processor mainframes are introduced, we may couple them together in a similar fashion. That is, whatever the current power of the fastest machine, we would like to be able to increase this arbitrarily by simple replication and coupling.

The justification for employing more parallel processors than are currently offered in a single integrated system is supported by our findings of the large degree of parallelism that exists in many scientific and engineering applications and their successful implementation on our earlier ICAP systems. This covers, to name a few, calculations in quantum chemistry, statistical mechanics, many engineering applications dealing with heat transfer and computational fluid dynamics, celestial mechanics, and fundamental algorithms used in all areas of science and engineering.

The "degree of parallelism" of a given piece of code can be characterized by three attributes: (1) how much of the computation can be run in parallel; (2) how often communication is required; (3) and how evenly the work can be divided across multiple processes. In the following section we consider analytically how these factors affect performance. Here we wish to discuss the proposed intersystem coupling of the ICAP/3090 system in this general framework.

First, there are many applications that are almost entirely parallel and—when partitioned across a number of processors—require very infrequent communication. Such applications are said to exhibit *coarse-grain parallelism*. For these we do not need a fast coupling and can use IBM 4.5-MB/S channels and channel-to-channel coupling to support interprocessor communication. Thus, our initial plan will be to couple all systems via channels with full point-to-point connectivity.

These applications are, however, more aptly described as subjects for distributed computing, rather

than for more communication-intensive parallel processing. For applications with smaller parallel grain size, i.e., characterized by more frequent communication, we require faster paths, because the overhead associated with communication can degrade performance severely.

Before discussing our requirements, it is important to define more carefully what we mean by the word "fast." In general, whatever the communication path, the time required to complete a transfer of information from one processor to another can be broken into two parts: latency and transfer speed. *Latency* defines the amount of time that must be spent in initializing the transfer and is (to a first approximation) independent of the *number* of bytes transferred. *Transfer speed* is characterized by hardware, in which (to first approximation) the amount of time spent varies linearly with the amount of data sent. For example, when we say that an IBM channel performs at 4.5 MB/S, we are specifying the transfer speed attribute. This is the asymptotic rate and is realized in practice only when the amount of data transferred is very large. For smaller transfers, the total time may be dominated by the latency, resulting in transfer speeds that are in the KB/S range. Latency may be broken again into two components: software path and hardware initialization. Using the IBM channel as an example, it is usually the software component that is dominant. *Software path*, in this case, implies interaction between the application and the operating system, through an interrupt to handle I/O.

Synchronous communication tends to ensure correctness of parallel execution.

The great majority of applications with *smaller parallel grain size* are characterized by frequent synchronizations and transfers of relatively small amounts of data. Overall, a single IBM channel operating at its peak rate would be satisfactory to serve in this capacity. However, given the large latencies involved, the resulting effective data rate is inadequate. Thus, to achieve faster coupling we will explore, through efforts within our laboratory, the use of two complementary approaches, both of which can have smaller latencies and have been used successfully in our previous ICAP systems (described later in this paper). The first approach employs shared memory, and the second utilizes fast message passing. Each serves different purposes, which we now consider.

In general, message-passing and shared-memory communication have the following characteristics. Message passing is most simply realized by a bus interconnect and the specification of synchronous communication protocols. Shared-memory communication is done asynchronously and is implemented in its simplest form by a multiported memory that is mapped into the same address space of the cooperating processors. Both communication methods have advantages and disadvantages. Synchronous communication tends to ensure correctness of parallel execution, because programs that use this mode are rigidly constrained by the data dependencies defined by message passing. The disadvantages are that a bus interconnect is limited by topological constraints, a tendency of the facility to become saturated when many processors attempt to use it, and also by substantial latencies because of the necessity for some type of handshaking protocol. Alternatively, a shared memory tends to have faster data-access rates, smaller latencies, and fewer connectivity restrictions. However, this requires more careful programming of accesses to guarantee program correctness, because implicit synchronization is lacking. ICAP/3090 will thus emphasize synchronization by message passing and data transfer by shared memory.

Two alternative ICAP/3090 systems are shown in Figures 1 and 2. In Figure 1, five 3090 Model 300s (each with three processors with vector attachments) are coupled by a large (several hundred megabytes) global shared memory and a fast-message-passing bus configured as a ring. Also shown is a full point-to-point connectivity implemented by channel-to-channel coupling. We expect that accesses to our shared memory will not be much slower than accesses to an expanded storage of any one of the 3090 clusters. In addition, we are investigating the possibility of permitting segments of this memory to support "read-modify-write" operations (such as test and set or fetch and add). In Figure 2, a similar system using Model 400s (four processor complexes) is illustrated. This simpler system differs from the

Figure 1 ICAP/3090 Model 300-based system

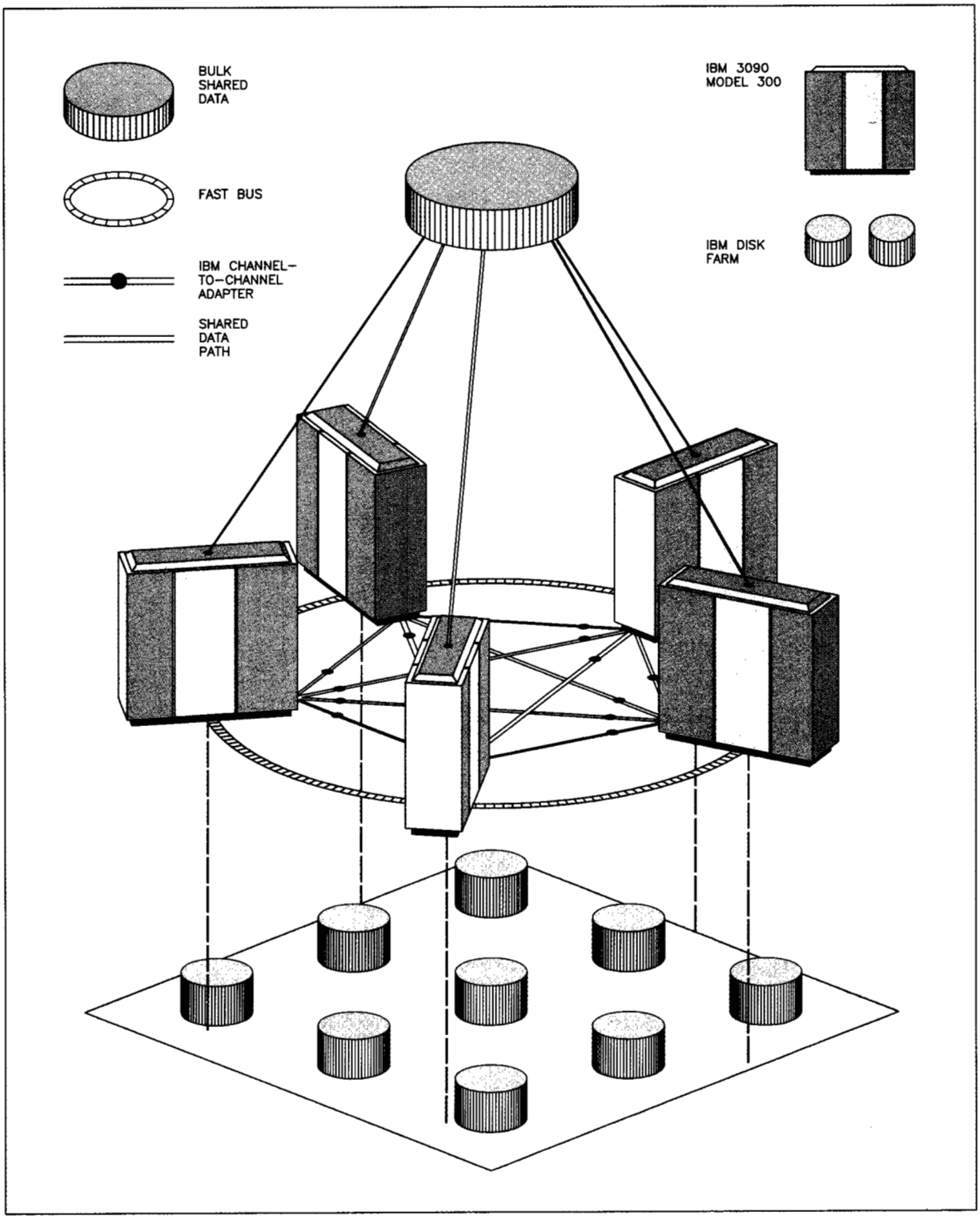
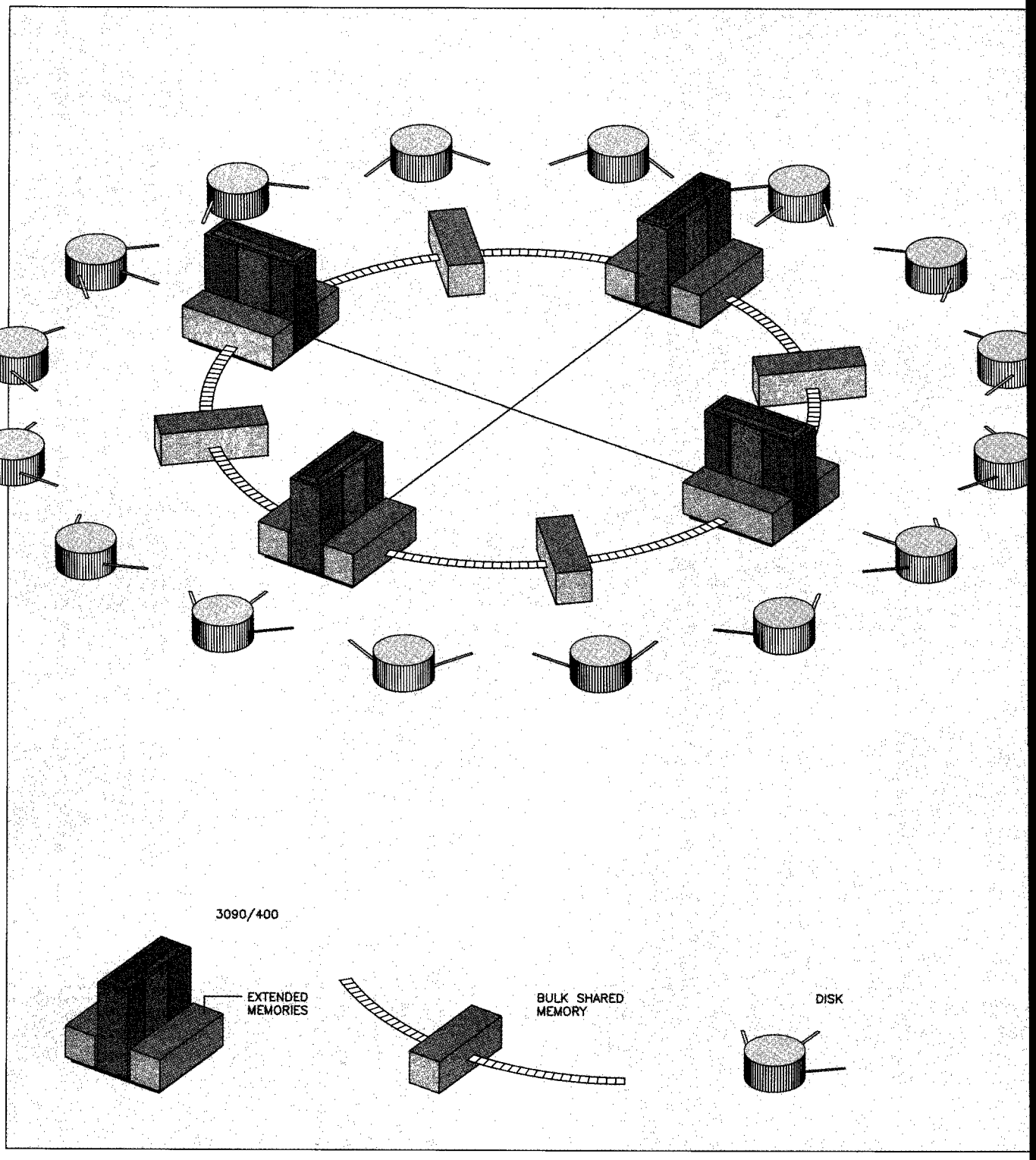


Figure 2 ICAP/3090 Model 400-based system



first in that multiple dual-ported shared memories are used to couple the clusters in a nearest-neighbor ring structure; further, it lacks a fast-bus message-passing capability. Also indicated in Figures 1 and 2 are large shared "disk farms" that can be implemented with existing technology. Disk farms serve as a secondary level of shared storage to greatly enhance the size of problems that can be addressed.

At present we have four IBM 3090s (Model 400s), each with vector capabilities and coupled by channels. Initial efforts in permitting multiple clusters to collectively work on a single problem, exhibiting large grain parallelism, have proven successful. To address smaller grain size problems we will require the development of shared memories (or fast buses) which are currently under study.

ICAP/3090 software. A discussion of parallel software must address two areas: operating systems and languages/compilers. From our experience with the earlier ICAP systems, we anticipate that we can extend the parallel operating systems software used on those systems to work on ICAP/3090. There, both the MVS and the VM operating systems were employed, using task-level parallelism in MVS and cooperating virtual machines (VMs) in the VM operating systems.[9] Of the two, VM seems to be the more expedient solution because of the potential ease of facilitating intersystem communication (via channels) between VMs, through either the Virtual Machine Communication Facility (VMCF) or the Inter User Communication Vehicle (IUCV). For a more detailed discussion of these issues, see Reference 10.

Communication through the bus and shared memory will require extensions to operating-system functions, although it is clear that latency effects must be minimized to achieve acceptable performance. The incorporation of these paths will proceed in an evolutionary manner as we explore the capabilities that our channel connections allow.

The language/compiler software is intended to be a superset of the existing Parallel FORTRAN (PF) software developed for single-system parallel programming. Reference 11 gives a comprehensive description of the PF software; in the following, we highlight several features of this software.

The PF compiler can be used to automatically parallelize code for multiple processors in which specific attention is paid to DO loops. Here, single or nested DO loops are analyzed to determine whether the code is parallelizable; if so, a cost analysis is performed to determine whether it is profitable to do so. In the event that the analysis is hindered by run-time considerations, compiler directives may be inserted in line to aid in this analysis. In this regard the compiler is similar to vectorizing compilers.

Parallel algorithms are critical for parallel engineering calculations.

Alternatively, the PF software offers a rich set of extensions to the FORTRAN language that permits the programmer to control the parallel operation of his code. The two constructs that are used most frequently are the following. The first, called PARALLEL LOOP, is used to implement parallel loops that have the additional abilities (beyond those of the automatic parallel loops) to define private variables for each processor and permit the creation of critical sections within the loop. The flexibility offered by these features makes this construct ideal for parallel algorithm development. Parallel algorithms are critical for parallel engineering calculations, where partial differential equations are typically solved by finite-difference or finite-element techniques.

The second construct is one for defining and executing parallel tasks. Tasks take the form of subroutines. A parallel program using this ability defines a master task that forks off a number of parallel tasks and upon their completion collects or joins the results, prior to perhaps more fork/join processes. Communication is achieved through subroutine argument passing as well as COMMON blocks and synchronization through locks and events.

PF may be used without modification for multiple clusters of 3090s when we employ extensions of our initial ICAP software to support intersystem communication through shared memory. For example, parallel loop implementation may be achieved by dedicated counters in shared memory that are ac-

cessed with mutual exclusion. Because we must do this as efficiently as possible, it is important that locked access be achieved on a hardware basis rather than through software spin locks. This implies a set of dedicated registers that support read-modify-write operations (such as fetch and add or test and set).

Alternatively, if we wish to use fork-and-join parallelism, we must allocate to one particular processor the role of master. Thereafter, forks can be achieved by synchronizations over the fast bus to user-defined interrupt handlers that are "spinning." Then, depending on the data transferred, these handlers can branch to the appropriate parallel tasks. In this scenario, the master serves and collects all data through the shared memory.

However, the development of the ICAP/3090 system is at a very early stage, from both hardware and software viewpoints. Before describing early ICAP systems, we digress briefly to discuss general and important performance issues.

Parallel processing performance issues

The fundamental reason for using parallel processing is to speed up the elapsed time of an application. We mentioned in the preceding section that the best performance is bounded by the degree of parallelism that our code exhibits, characterized by three attributes: (1) the fraction of the code that is parallelizable, (2) the amount of communication required, and (3) load imbalancing when we partition the problem across multiple processors. In fact, while we list these as separate factors, they are all interrelated. For simplicity, however, we consider each factor separately, pointing out, where appropriate, the subtleties that exist.

Fraction of parallel code. The most fundamental factor in determining parallel performance is the fraction of the code that is parallel. We define this fraction to be that of the total time required for the job. Consider a problem with that fraction equal to X. The remainder, $1 - X$, is sequential. Suppose that we have P processors over which the parallel portion can be distributed equally without any overhead. The sequential portion can be executed by only one of the P processors. Then the best we can expect, compared to running the entire problem on only one processor, is realized by a speedup factor S_P of

$$S_P = \frac{1}{1 - X + X/P}. \tag{1}$$

If X is unity, i.e., all of the code is parallel, then the speedup is equal to the number of processors (linear speedup). Otherwise, this limiting expression, known as Amdahl's law, applies. It generalizes, beyond the narrow concern of parallel processing performance, to describe the limiting behavior of any system that can exist in either of the following two fractional states: one that "performs" at a rate of unity (or normalized to unity), and the other at a normalized rate of P. For example, if we are describing the possible enhancements of vector processing compared to scalar processing, the same expression holds, except that in this case P means how much faster the vector processing is, and X is the fraction of code that is vectorizable.

It is evident that the application of Amdahl's law results in speedups that are very sensitive to the fraction of the code that is parallelizable. Shown in Figure 3 are speedups, for differing values of X, as a function of P. As X becomes smaller, speedups more rapidly approach asymptotic values near which the addition of more processors is not cost-effective. This observation is the one most frequently cited in negative assessments for the prospects of massive parallelism (i.e., the use of massive numbers of processors).

However, the situation is a little more subtle than this. First, how do we determine the fraction X for an application? Consider a simple example of solving a system of N linear equations involving N unknowns. We may write this problem in matrix form as follows:

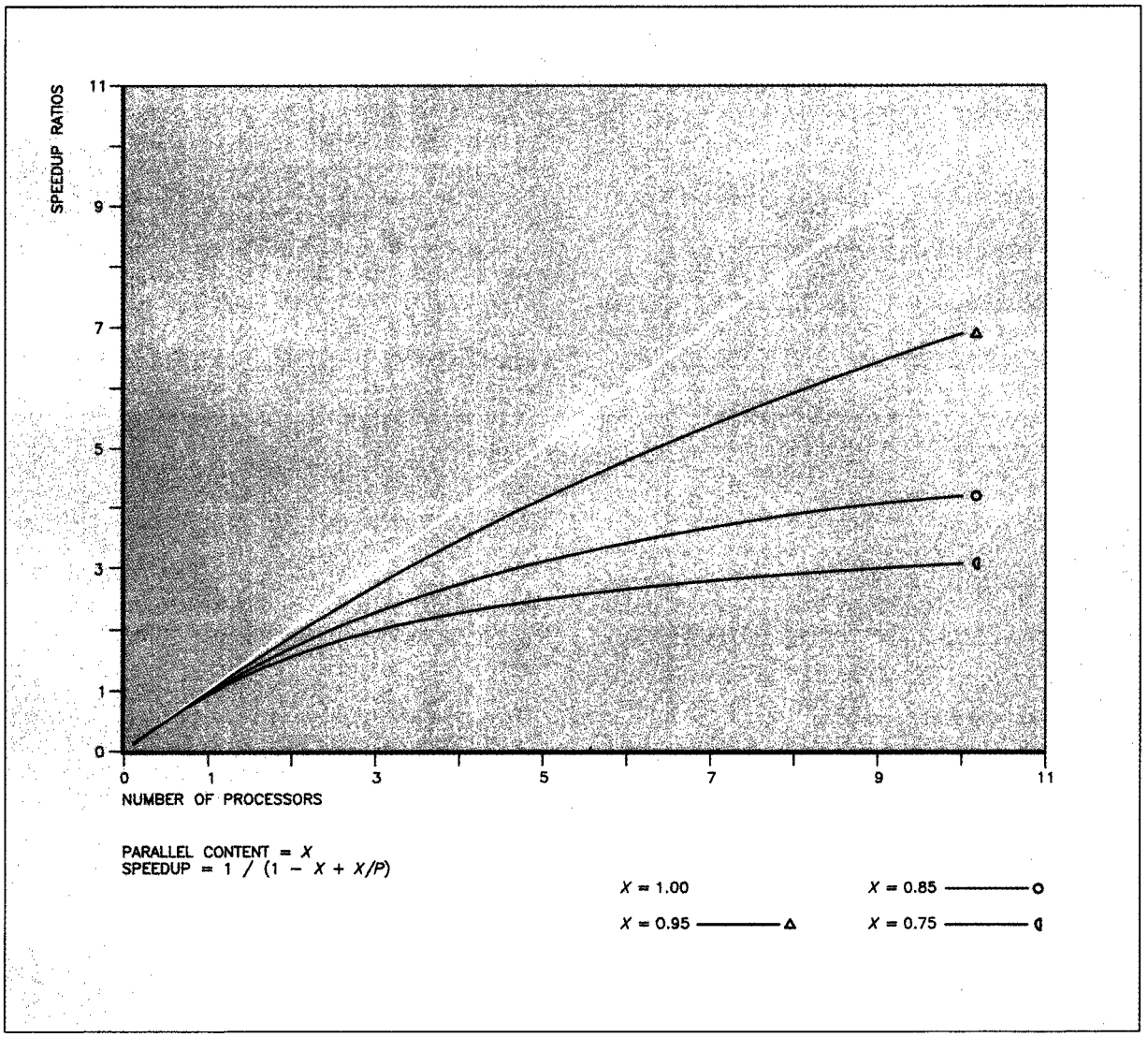
$$\mathbf{A} \times \mathbf{y} = \mathbf{b}, \tag{2}$$

where $\mathbf{A}$ is a matrix of size N by N, $\mathbf{y}$ is a vector of N elements that we wish to find, and $\mathbf{b}$ is a vector of N elements that is given as the right-hand side of our problem. Then, if the construction of $\mathbf{A}$ is nominally the only sequential part of the problem (e.g., it is read in from some sequential device), the amount of sequential time spent is of the order of $N^2[O(N^2)]$. Assume that the solution phase is parallelizable, and, if we use Gaussian elimination, this takes $O(N^3)$ operations (multiplications and additions) or time. Then the fraction of the code that is parallel is the following:

$$X = \frac{rN^3}{rN^3 + sN^2}. \tag{3}$$

Here we have identified the coefficients (r and s) that give the exact correspondence to actual time instead

Figure 3 Speedup constraints of Amdahl's law



of the vague "order-of" expression. The important point is that as N becomes large, the fraction of this application that is parallel approaches unity. Thus, it is not sufficient simply to quote Amdahl's law as predicting whether parallel processing supports massive parallelism. Problem size and the dependent relationship of the sequential and parallel part determine how many processors may be gainfully employed.

However, this example has been oversimplified. The solution of our set of simultaneous equations is not perfectly parallel, because we must consider the degrading effects of load balancing and communication. Considering load balancing, it is clear that the assumption that the parallel part is equally divisible for all values of P ignores the reality that any computation consists of a fixed number of discrete operations and cannot be so divided. This may be

further compounded when we consider in more detail the structure of the application and the role that communication can play in defining the fraction of parallel code.

Because of the structure of an application, it is possible that the fraction of parallel code can change as the job proceeds. For example, considering our solution of a set of linear equations, it is true that the total computation complexity is $O(N^3)$. However, this total amount comprises the execution of N separate parallel stages, each of complexity $O(m^2)$, where m starts at N and goes to 1. Each stage must be completed before the next can begin. This imposes the need for synchronization and possibly the passage of data between processors. Both add an effectively sequential time of $O(P)$ or $O(m * P)$ before the present stage can begin, and thus each stage has associated with it its own parallel fraction. This fraction is initially large and becomes progressively smaller as the solution proceeds. This implies that one can start with a large number of processors but thereafter discard them appropriately, until it makes sense to finish with only one processor. So here we have a (not atypical) problem that cannot be characterized by a single parameter of parallel-fraction content.

Another subtlety that may occur is that, for a fixed problem size, the fractional content X may depend on the number of processors used. To illustrate this, consider the problem of adding N numbers. Whereas it is not necessarily optimal in terms of minimizing the number of concurrent elementary operations involved, the most straightforward way of doing this is to allow each processor to compute a partial sum (of approximately the same number of elements) and give to one of these processors the remaining task of adding up the partial sums. The first part is the parallel portion and the latter the sequential stage. The fraction of parallel content of the algorithm is thus approximately

$$X = \frac{N - P}{N}. \tag{4}$$

If we substitute this expression in Equation 1 and plot the speedup as a function of P, we find that there exists some critical value of P beyond which the addition of more processors increases solution time. In this case, we have an analytic solution for the optimal number of processors, which is $\sqrt{N}$. Although this example may seem somewhat artificial, in practice a good many applications exhibit the property of having the sequential fraction increase with the number of processors.

In summary, the fraction of parallel code is the primary factor that determines whether parallel processing is profitable, and if so, determines the number of processors which is optimally cost-effective. In practice, it is found that the parallel content of most applications grows as problem size increases. Thus the answer to the question of the optimal number of processors depends on the size of the problem. However, this must be further refined in terms of the details of implementation and the structure of the calculation.

Communication. While the fraction of parallel code is the primary consideration in determining how profitable it is to use parallel processing, it is by no means the final consideration. This is because, as we have seen previously, we must account for the extra time required for our processors to communicate as this overhead increases the effective sequential fraction of the code.

The degradation that communication can cause may be seen, perhaps in a simpler fashion, by examining the expression for the total time spent in solving our problem with a set of P processors. If the parallel execution time to be partitioned equally among our P processors is T_p and the sequential time is T_s, the total execution time is given as follows:

$$T_{par} = T_s + \frac{T_p}{P} + f(P). \tag{5}$$

The extra term $f(P)$ represents the additional time spent in communication, and at minimum can be some constant or a positive function of the number of processors. By ignoring some of the subtleties discussed in the previous section (assume that T_s and T_p do not depend on P and to first approximation P divides T_p more or less evenly over a large range of P), the following account holds. If $f(P)$ is a constant, we can expect the minimum total execution time or maximum speedup to be asymptotically bounded. If $f(P)$ is a positive increasing function of P, the total time can be decomposed into two parts, one increasing and the other decreasing monotonically with P. In this case, we may expect that there is some optimal value of P for which the execution time is minimized and beyond which the total execution time actually increases as we add more processors. Using a little calculus, we may solve for this optimum value, P_{opt}, from the following equation:

$$P_{\text{opt}}^2 f'(P_{\text{opt}}) = XT_{\text{seq}}, \quad (6)$$

where X is the fraction of the code that is parallelizable and T_{seq} is the total time of our problem executed in sequential mode. $f'(P)$ is the derivative of $f(P)$ with respect to P. Solving for P_{opt} then provides the following straightforward algebraic equation for the maximum speedup:

$$S_{P_{\text{opt}}} = \frac{1}{(1 - X) + X/P_{\text{opt}} + f(P_{\text{opt}})/T_{\text{seq}}}. \quad (7)$$

To illustrate the effects that arise, consider the following example. Suppose we have an application that has a repeating structure wherein we first perform the parallel portion with P processors, and then one of these (call it the master) executes a sequential section. We require synchronizations between the $P - 1$ processors and the master before the sequential part can be done, and prior to the next parallel execution. The communication time for synchronization—given that serialization is implicit in one processor reading, receiving or issuing $P - 1$ messages or interrupts—scales as $P - 1$ and can be written

$$f(P) = a(P - 1) + b, \quad (8)$$

where the magnitude of a depends on how the synchronization is done (e.g., shared memory, bus, or network) and b is a latency term that is included for completeness. Using this expression, the optimal number of processors (using Equation 6) is given as follows:

$$P_{\text{opt}} = \sqrt{XT_{\text{seq}}/a}. \quad (9)$$

This simple example shows that the optimal number decreases quadratically as a function of decreasing communication speed. Thus it is imperative that communication be achieved as rapidly as possible. This effect can be made less restrictive by increasing the problem size (T_{seq}). Additionally, because X usually increases with problem size, this beneficial effect is multiplicative.

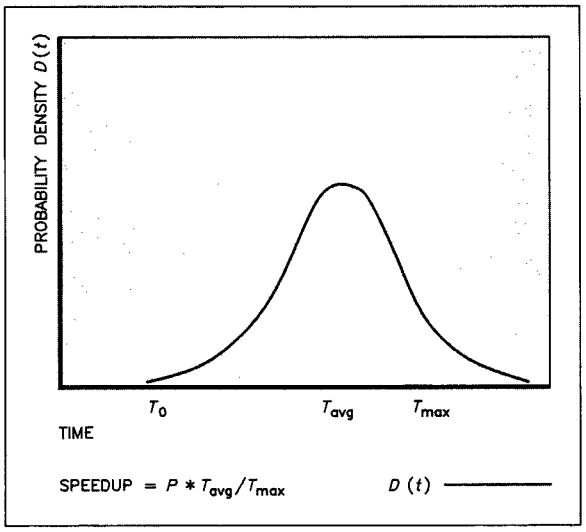
This simple model may be extended to include more complicated forms of communication, e.g., synchronization followed by data passing, where the quantity of data passed may in turn be a function of P. The model may also be extended by redefining the optimal number of processors, not simply in terms of minimizing the total elapsed time, but rather in a more realistic cost-effective sense, by solving for the maximum number of processors beyond which some marginal increment of additional speedup is not met. (In the example previously cited, this leads to a complicated quartic equation for P_{opt}.) However, it is not our intent to explore all the complex derivations that can occur. Also, as discussed in the previous section, the fraction of the code that is parallel (i.e., X) may also be a (decreasing) function of the number of processors employed. This further complicates matters and makes the above analytic optimality expressions the upper bounds on performance. The important point is that the performance degradation can be nonlinear, as we impose a greater connectivity and communication between cooperating tasks.

The performance degradation can be nonlinear, as we impose a greater connectivity and communication between cooperating tasks.

Load balancing. The basic goal of parallel processing is to keep all processors busy all of the time. Deviations from this ideal are called load-balancing problems. It is clear that load balancing is faced with the following dilemmas. First, for a given problem size, there is only so much work to be performed, and thus we are limited to some maximum number of processors. Second, because of the discrete nature of any parallel calculation, the division of labor is more evenly balanced for certain numbers of processors than others. These constraints may be compounded if, as suggested previously, the amount of parallel work changes as the job proceeds and communication is required.

However, additional complications may arise. Thus far, we have been implicitly describing a single-user parallel processing system and applications that consist of a fixed number of operations. This allows us a tractable static determination of the number of processors to use. If either of these conditions is not met, we must look more carefully at load-balancing degradation.

Figure 4 Statistical parameters that govern performance in time-shared systems



The basic aspect of parallel computation that we wish to examine is what happens when the elapsed time to execute a task is not precisely determined, i.e., execution time is randomly distributed. This situation typifies the elapsed time of a job executed by a general time-shared system wherein large numbers of jobs compete for resources. The heavier the demand for system resources, obviously the longer the turnaround time will be for a sequential job. If, however, we wish to determine the performance of a parallel job in such an environment, the degradation to performance (i.e., elapsed time) can become more complex. We illustrate this with the following rudimentary example.

Consider a job that is entirely parallel and is executed on a P-multiple-processor, time-shared system. Assume further that we may partition the job into P equal tasks that require no communication and assign them to specific processors. In the absence of any other jobs in the system, each task finishes at an identical point in time, and the speedup is equal to P (linear speedup). Now consider the effects of a number of other jobs competing in an unbiased manner for processors. The result is that the time for the completion of any one of our tasks will be variable. More precisely, we can say that the time for the completion of any task is randomly and identically distributed and is characterized by a probability distribution. This distribution is lower-bounded by the time the tasks take in the empty system. The width of the distribution depends on many factors, including the dispatching policy of the operating system, the number of jobs competing for processors, and the job service and arrival time distributions.

The best that we can do in predicting performance under these conditions is to determine the average or expected speedup. To do this, we need to estimate two quantities. The first is the average time T_{avg} at which the tasks complete. The second, T_{max}, is the expected value of the time for the last task to complete. The time for the last task defines when our parallel job is finished. With these two quantities, the following equation defines the average speedup of our parallel job relative to the sequential job executed in the same environment:

$$S_P = \frac{PT_{avg}}{T_{max}}. \tag{10}$$

To see this, it is necessary to observe that the ratio of Equation 10 is simply the fraction of the processors that are busy during the total job completion time. Equivalently, we may recognize this as simply the ratio of the expected elapsed times of the sequential-to-parallel implementations. Here, the expected sequential time is estimated as the sum of the expected times of running the P tasks on one processor. Thus, to predict performance, we must be able to evaluate these two parameters. We can do this if we know our underlying probability distribution.

Assuming that we know this distribution $D(t)$, the estimation of average time T_{avg} is straightforward. It is simply the average of our distribution and is indicated for a sample distribution in Figure 4. Similarly, the expected maximum T_{max} can be estimated using a theory of statistics known as "order statistics." Typically, this theory is used to study sampling problems. In our case, we are sampling from a distribution P times and we wish to estimate the expected maximum value we will observe. One approximation to this value is a quantity known as the "characteristic maximum." This is calculated by performing a definite integral over the distribution from zero to some upper limit and requiring that the result equal the fraction $(P-1)/P$. The upper limit is the characteristic maximum (Equation 11), as illustrated in Figure 4 for a case in which P is equal to 8:

$$\int_0^{T_{max}} D(t)\, dt = \frac{P-1}{P}. \tag{11}$$

It is clear that the average speedup exhibits the following dependencies. As more jobs are introduced, competition for processors increases. This may increase the width of our underlying probability distribution. As width increases, the difference between the average and expected maximum times increases, resulting in poorer performance and lower speedups for a parallel application using P processors. Increasing the number of processors over which our application is partitioned is subject to two competing influences. On the one hand, the underlying distribution tends to be narrower and shifted to shorter times (because the amount of work is less in each parallel task). However, the expected maximum increases (because the integration is over a distribution to a fraction that is closer to unity). Whether performance is enhanced or degraded depends in this case upon the relative magnitudes of these effects.

The motivation behind ICAP was initially limited to solving large problems in theoretical chemistry.

This method of estimation of speedup is overly simplistic, however, because the assumption that the underlying distribution is static may be invalid. That is, the method ignores the fact that as some of our parallel tasks complete, the total number of jobs in the system decreases. Similarly, a more accurate prediction of speedup would require that we estimate the expected elapsed time of our sequential job from a probability distribution that takes into account that there are $P - 1$ fewer jobs in the system than under our parallel scenario. Thus our estimation that the sequential time is P times T_{avg} (the numerator in Equation 10) is too large, and the predicted speedup is overly optimistic.

In summary, this analysis is meant to show only that load-balancing issues that arise in time-shared parallel processing systems can give rise to additional performance degradation beyond that due to the fraction of parallel content, communication, and static load-balancing problems. Although it is somewhat theoretical and cannot be used in a simple fashion to predict exact performance (because the exact nature of the underlying distributions is generally unknown), the analysis points out the essential statistical factors that influence performance.

Early ICAP systems

To introduce the earlier ICAP systems and their influence in defining ICAP/3090, it is appropriate to begin by discussing the applications that led to their design. The motivation behind ICAP was initially limited to solving large problems in theoretical chemistry. This area includes quantum chemistry calculations employing self-consistent field methods, and statistical mechanics calculations using Monte Carlo and molecular dynamics methods. Although it is not our intent here to go into detail about these areas, several comments might be helpful.

The reason for performing such calculations is to gain insight into the behavior of matter. Quantum chemistry calculations can be used to understand the properties of isolated molecules, i.e., their energy states, spectroscopic properties, etc. Alternatively, they may be used to describe the interaction of one molecule with one or more other molecules, i.e., two- or many-body potentials. Statistical mechanics calculations deal with large ensembles of molecules and can be used to predict the thermodynamic properties of gases, liquids, and solids, such as heat conductivity, free energy, etc. An essential ingredient for statistical calculations is the intermolecular potentials that may be calculated *ab initio* by quantum chemistry theory. Thus there is a natural and symbiotic relationship between the two fields. In terms of parallel processing, all applications share the following important properties.

The first attribute of these calculations is that the parallelism within each is obvious. For example, considering a molecular dynamics calculation, the problem consists in studying the time evolution of an ensemble of particles for which we know an expression that describes how one body attracts or repels another, e.g., a two-body potential. Then, at each time step, we wish to calculate the total force on each body and move it to a new position on the basis of this force. The parallelism is evident in that the forces on each body can be calculated in a cumulative parallel manner as each processor or process calculates the independent two-body terms.

All that needs to be done at the end of this period is have a synchronization followed by a global addition of all contributions from each processor.

The selection of master and slaves was made from those available at the time the project began in early 1983.

Again considering the same example, the second attribute concerns the complexity of each individual two-body computation. A long-range two-body potential, derived from quantum chemistry calculations for some complex molecular species, is typically a very complicated expression. It therefore requires a large amount of time to perform all the two-body calculations before the final global sum is performed, followed by the projection of the new positions. This is even more true when we consider potentials for three-or-more-body interactions. In short, we are describing what has come to be known as an application with very-large-grain-size parallelism. This implies a great deal of computation prior to any required communication between processors.

A third attribute, inferrable in part from the discussion just given, is that the amount of code that is parallelizable is very close to 100 percent. The only sequential section involves the global summation and the prediction of the new coordinates. Thus, the limitation imposed by Amdahl's law on the number of processors that may be gainfully employed in parallel computation is not severe for these calculations. That is, for problem sizes of interest, tens to hundreds of processors can be employed before the sequential portion of the calculation begins to dominate the total solution time. Moreover, load balancing is typically easy to achieve. The most important constraint is the minimization of the overhead that arises in communicating partial forces and broadcasting new positions at the end and beginning of each time step. This additionally limits the number of processors we may employ.

Given these characteristics of our applications, it was obvious what type of parallel computer we could assemble to perform these types of calculations. All that we required was an elementary master/slave system in which the slaves would, for example, calculate the two-body contributions at each step and send the results to the host or master. The host would then add them up, predict the new positions, and send these back to the slaves for the next period of computation. The example given has thus far been that for a typical molecular dynamics calculation, but analogous operations also pertain to the remainder of the applications. We now describe the architecture of the initial ICAP system.

Architecture. Because we wanted to assemble such a system in a short period of time and move toward the main goal of solving large problems, the selection of master and slaves was made from those available at the time the project began in early 1983. The slaves or attached processors (APs) had to satisfy the following requirements. Because of cost, space, and complexity of the overall system, the total number of attached processors was limited to approximately 10. This demanded—given our goal of supercomputing performance—that an AP be a powerful processor in its own right and have 64-bit precision floating-point hardware, as was required by our applications. Additionally, each AP needed a real memory of the order of a megaword (64-bit words) and several hundred megawords of disk space. Given these considerations, the optimal choice was a Floating Point Systems Model 164 (FPS-164).

The FPS-164 has a peak performance of 11 million floating-point operations per second (MFLOPS) and comprises multiple functional units, including a floating-point adder and multiplier, that may be executed during each machine cycle. Up to nine other operations may additionally be performed during each machine cycle (182 nanoseconds), including memory fetch and register-to-register transfers. The FPS-164 is supported by a number of FORTRAN cross-compilers for a corresponding number of host machines. These compilers do a good job of producing optimized object code that packs many instructions into a given macro- or long-word-length instruction. The floating-point units are pipelined and support chaining with one another. The machine has a 24-bit addressing capability and supports standard I/O to its own disk system.

The choice of host was based on the following considerations. First, because data transfer between slaves and host represents overhead in our parallel computations, we required superior transfer rates to

the extent possible. Second, we required a host that had superior I/O capability and offered an easy-to-use operating environment. On the basis of these considerations, an IBM 43XX mainframe was selected. For example, a dyadic 4381 can have up to 24 independent channels, any number of which can be coupled to a corresponding AP. Each channel can operate at a maximum transfer rate of 3 megabytes per second (MB/s) and the channels can be driven in parallel with one another. For ease of use, the 43XX offers the Virtual Machine (VM/SP) operating system[12] for which FPS offers all the intersystem support software as a standard product.

With these choices, the system was incremented gradually over approximately two years to include 10 FPS-164s coupled to two IBM hosts, as shown in Figure 5. As indicated, one host was a dyadic IBM 4381 that could couple to up to all ten APs, and the other host was an IBM 4341 that could couple to up to three APs. The 3088 switching units allowed the latter three APs to be configured to either host. Additionally, a third IBM 4341 was placed in the system and served as a graphics station attached to a number of graphics terminals. All of the 43XX systems had channel-to-channel coupling. Finally, large amounts of external storage in the form of IBM 3350 and 3380 disk systems were included, totaling approximately 25 gigabytes systemwide. This system was called ICAP/1.

In parallel with the development of ICAP/1, we assembled a similar but more powerful ICAP system called ICAP/2. Like the ICAP/1 system, it was a master-slave system, but it differed in that the host was a single dyadic IBM 3081 that coupled to a number of the Floating Point Systems FPS-264s. The FPS-264 is very similar architecturally to the FPS-164, but it is approximately three to four times faster, with a cycle time of 52 nanoseconds. Again, IBM 3-MB/s channels were employed to couple host to slaves. Each slave had its own disk subsystem (each with approximately one gigabyte), and the host a separate large complement of disk storage. Eventually, as with ICAP/1, this system included ten attached processors. One further distinction between the two systems was that the IBM 3081 ran under the IBM MVS operating system[13] rather than the VM/SP operating system.

Both ICAP/1 and ICAP/2 were essentially single-user systems in the following sense. A parallel job initiated on a host was assigned a number of attached processors that would be retained until job completion. There was no dynamic switching of a given attached processor among multiple parallel jobs. This was consistent with the main goal of the system: performance of large-scale production calculations. It also eliminated from consideration the performance degradation of statistical load imbalancing previously discussed. One exception, however, was in effect for ICAP/1. The three processors attached to the IBM 4341 permitted AP allocation and deallocation among multiple parallel jobs and thus served as a tool for parallel program and algorithm development prior to migration to the production environment.

Table 1 Applications performance on ICAP/1

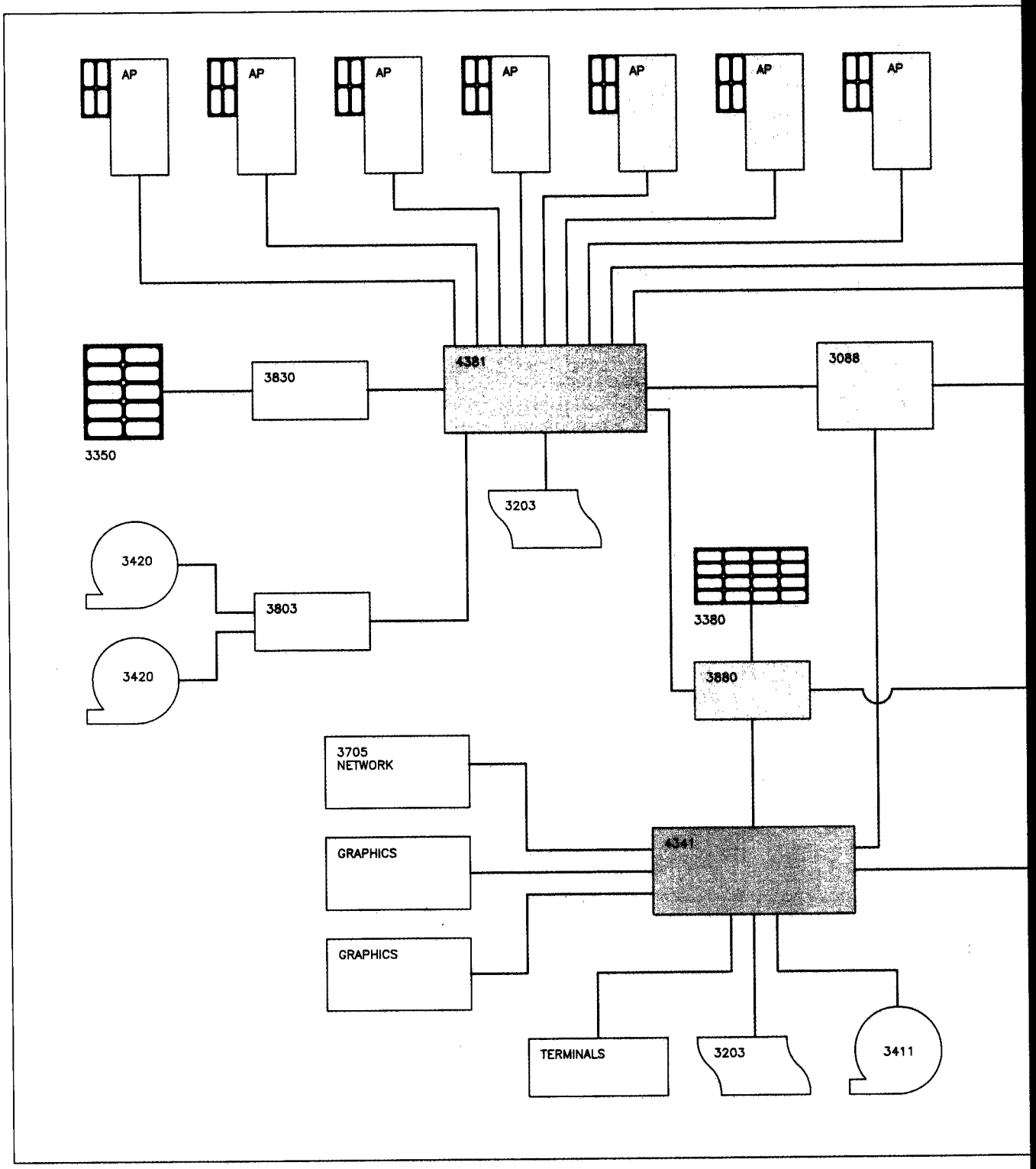
Job	Elapsed Time for ICAP/1 (minutes)				
	1 AP	3 APs	6 APs	10 APs	CRAY-XMP
Integrals (27 atoms)	71.7	24.0	12.3	7.8	7.6
SCF (27 atoms)	25.2	9.4	5.9	4.9	3.6
Integrals (42 atoms)	203.7	68.9	38.3	21.2	23.2
SCF (42 atoms)	73.0	26.0	14.3	10.6	8.7
Monte Carlo	162.1	57.8	32.0	22.0	20.4
Molecular dynamics	99.6	34.6	19.3	13.7	17.0
Seismic	33.8	11.8	6.6	4.3	5.6

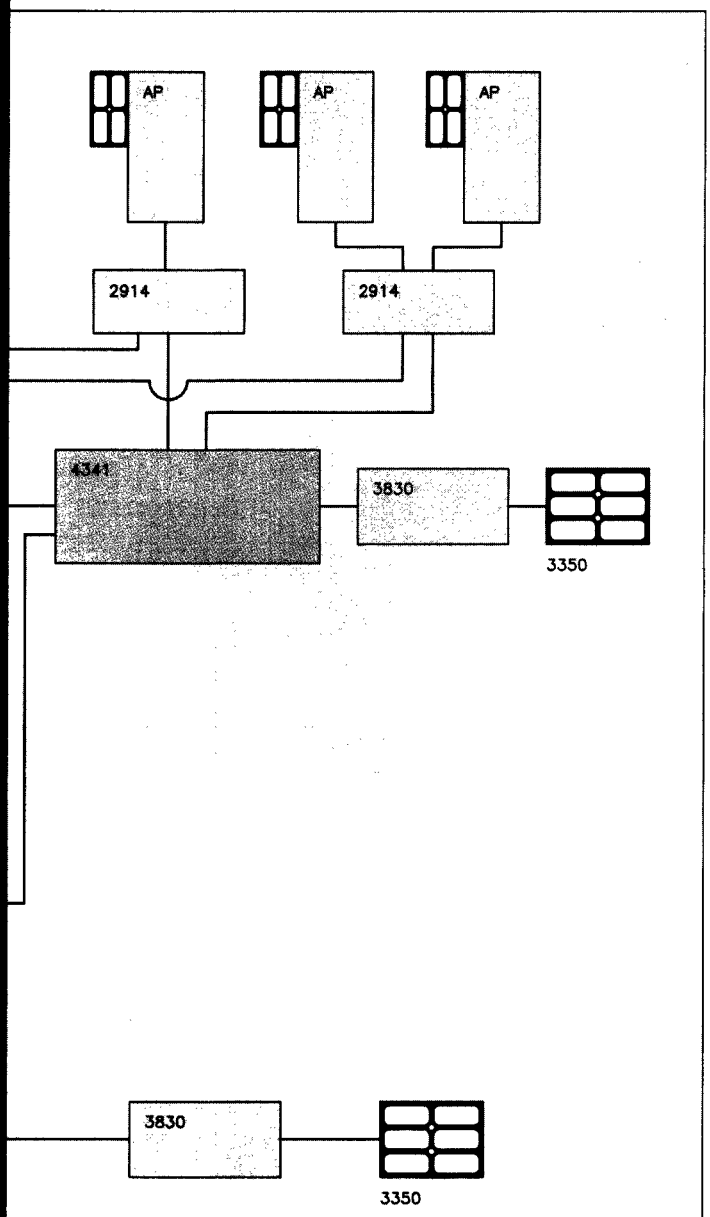
Table 2 Applications performance on ICAP/2

Job	Elapsed Time for ICAP/2 (minutes)				
	1 AP	3 APs	6 APs	10 APs	CRAY-XMP
Integrals (27 atoms)	19.1	6.5	3.3	2.3	7.6
SCF (27 atoms)	10.6	5.2	3.7	3.4	3.6
Integrals (42 atoms)	55.0	18.7	9.3	6.1	23.2
SCF (42 atoms)	24.1	9.1	5.6	4.7	8.7
Monte Carlo	60.0	20.9	11.4	7.7	20.4
Molecular dynamics	29.6	10.6	5.9	4.2	17.0

Performance. The initial goal of achieving supercomputer performance was achieved almost immediately. This is indicated in Tables 1 and 2, where elapsed times for a set of applications are compared on ICAP/1 and ICAP/2, respectively (as a function of the number of slaves used in parallel) versus a single-processor CRAY-XMP. The applications cited are an integrals-generating program used to construct matrix elements for a self-consistent field (SCF) calculation, for the iterative SCF code, and for illustrative many-body-interaction molecular dynamics and Metropolis Monte Carlo codes. It is important to note that the codes were not optimized for any of

Figure 5 Early, loosely coupled array of processors (ICAP/1)





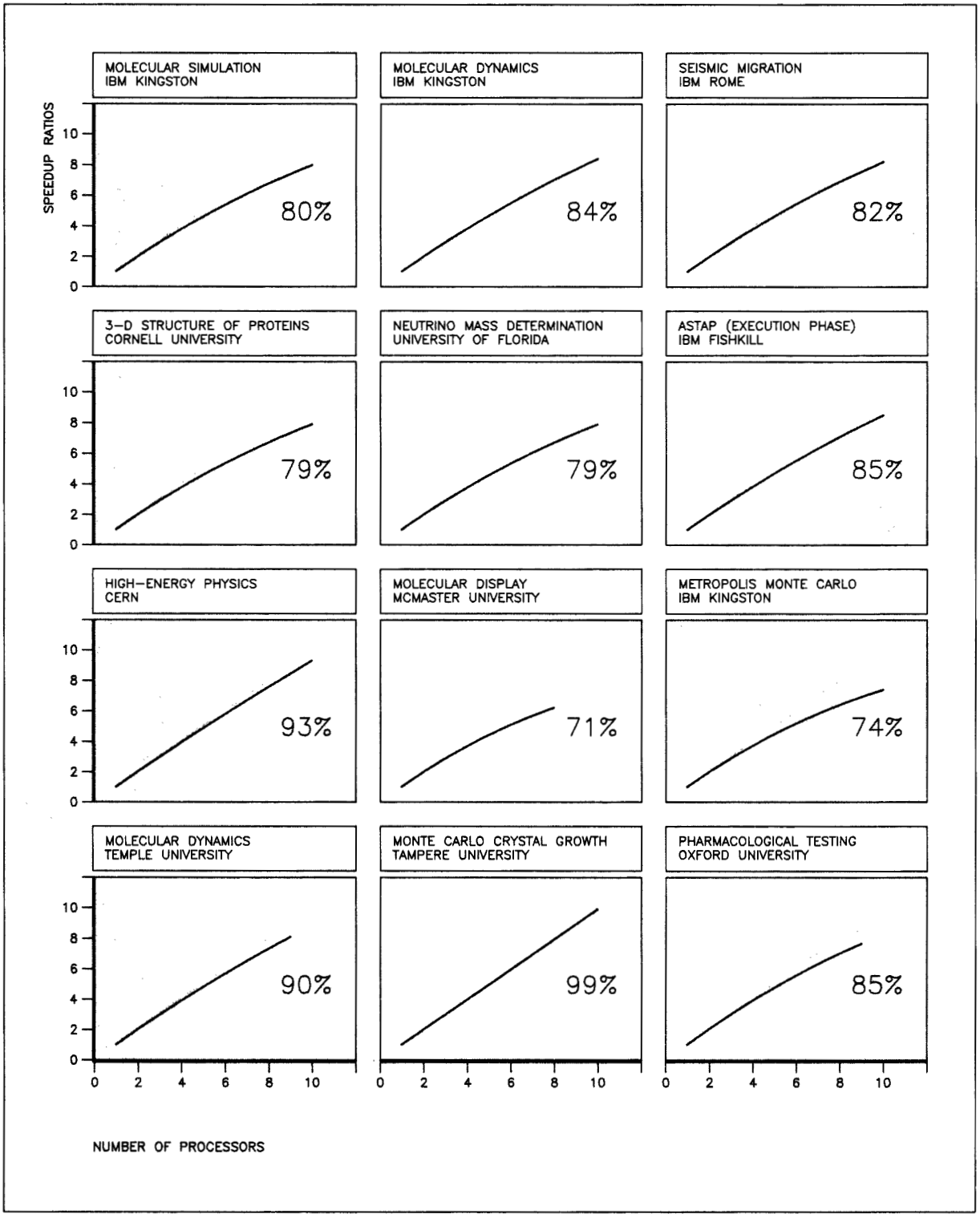
the systems, and thus these measurements cannot be used as anything like a definitive comparison. However, they do serve to indicate roughly the aggregate computing power of the machines for these applications.

With this qualification, Tables 1 and 2 show that for these applications, ICAP/1, with eight to eleven processors, and ICAP/2, with three to four processors, each roughly equal the computing power of a single CRAY-XMP processor. No data were available for comparison of these jobs running in parallel on an XMP.

The success in applying ICAP to these application areas was attributable to their large parallel grain sizes. The amount of time spent computing was large compared to the amount of interprocessor communication required. A representative but more detailed description of a number of parallel applications in theoretical chemistry executed on the ICAP system may be found in Reference 14. This characteristic was found to apply to many other applications, including the analysis of high-energy physics data, the determination of protein structures, seismic migration analysis, and circuit simulation. Typical performance results are indicated in Figure 6 for several such applications. Here the speedup factor is plotted versus the number of parallel attached processors used. *Speedup* is defined as the ratio of the times a given application took to run on P processors compared to that running sequentially on one processor.

These examples include (going from left to right, top to bottom, and indicating, where appropriate, collaborators and their affiliations) molecular energy determinations, molecular dynamics studies of water, oil reservoir seismic analysis, predictions of protein structure (Professor H. A. Scheraga at Cornell University); determination of the wavefunction for HeH^+ employed in predicting the mass of the neutrino (Professor W. Kolos at the University of Florida); electronic circuit simulation,[15] high-energy physics data analysis (Drs. F. Carminati, R. Mount, H. Newman, and H. Pohl at CERN); chemical reactivity determination through electron density calculations (Professor R. F. W. Bader at McMaster University, Ontario, Canada); stochastic simulations of fluids, study of the liquid–solid silicon interface (Dr. E. Gawlinsky at Temple University), silicon crystal growth (Professor K. Kaski at Tampere University of Technology, Finland); and the indexing of biologically active chemicals (Professor G. Richards at Oxford University, England).

Figure 6 Speedup for typical large parallel grain size applications



However, the ICAP systems did not do as well for applications characterized by medium- or small-grain parallelism. That is, the burden of interprocessor communication through a host intermediary using 3-MB/S channels was often too great. This was the case particularly for engineering applications, where partial differential equations are typically solved following standard discretization techniques such as finite differencing or finite elements. Here, the basic algorithms[16] that are required to solve large linear sets of equations or that perform matrix diagonalization are not characterized by large parallel grain sizes. These algorithms require more efficient interprocessor communication. This concern gave impetus to attempting to improve communication on ICAP.

Communication. The extensions made to the initial ICAP systems dealt primarily with enhancing communication. Two hardware additions sought to achieve this.

The first was the inclusion of a number of shared memories developed by Professor Martin Schultz of Yale University and Scientific Computing Associates (SCA). These memories were fast solid-state disks that were coupled to the I/O ports of the FPS machines and were directly addressable by the attached processors. This configuration is illustrated in Figure 7 for the ICAP/1 system; ICAP/2 was identically extended. Shown are five memories, each multiplexed four ways and each being 32 MB in size, linked in a double-ring structure around the ten processors. Each processor has two independent paths to a separate memory. A processor transmits data to or from a memory at a nominal peak rate of 44 MB/S on ICAP/1 (one word per machine cycle) and 38 MB/S on ICAP/2 (one word every fourth machine cycle). Additionally, one large, bulk-shared memory developed by SCA of size 512 MB (sectioned into four independent banks of 128 MB and multiplexed 12 ways) was coupled to all ten processors. The 12-way multiplexing was distributed by three independent buses, each capable of peak transfer rates of 44 MB/S on ICAP/1 and 38 MB/S on ICAP/2. The peak aggregate transfer rates (132 MB/S on ICAP/1 and 114 MB/S on ICAP/2) were realizable if three processors using different buses addressed separate memory banks. Software to employ these memories was developed by SCA[17] and extended within our laboratory to include functions such as synchronization between processors and locks supporting critical sections.

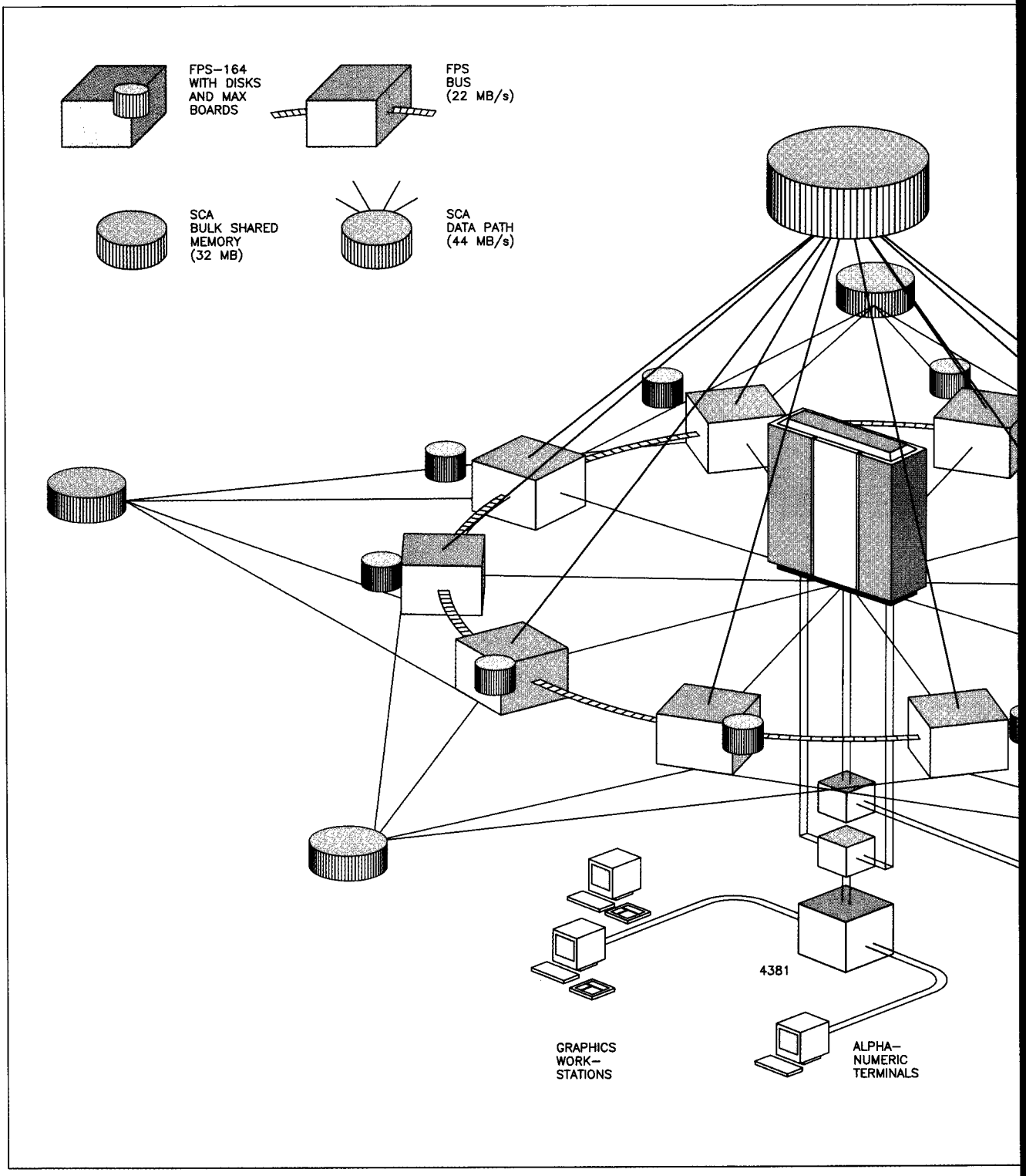
The second hardware addition to the ICAP clusters was a 32-bit-wide fast bus (built by Floating Point Systems) that linked all attached processors. This is also illustrated in Figure 7, showing each processor attached to a node on the bus. The nominal peak speeds were 22 MB/S from AP to node and approximately 32 MB/S between nodes, for both ICAP/1 and ICAP/2. Software to use this facility was developed by FPS[18] and included the ability for any non-nearest-neighbor communication, as well as for broadcasting from one to all.

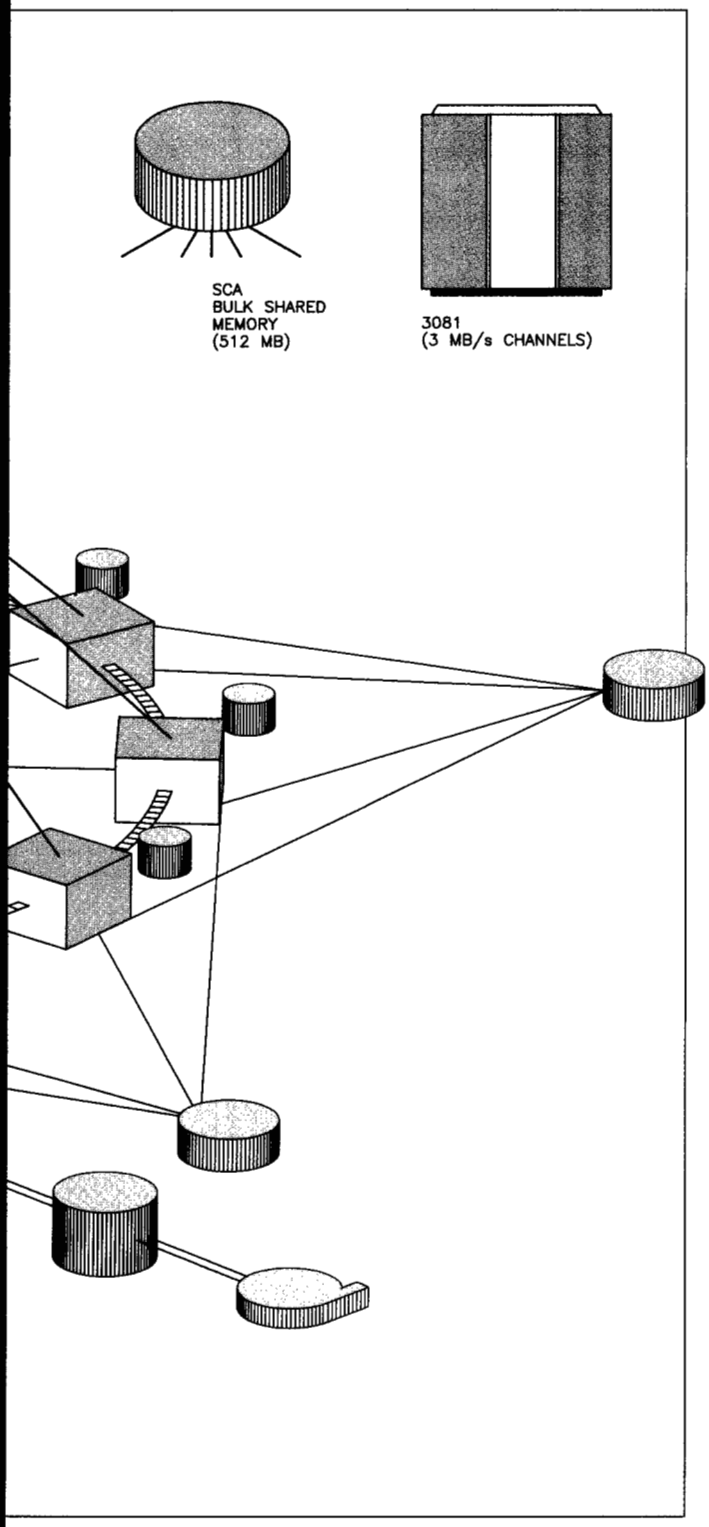
The partitioning of a parallel program and its communication is dictated by operational and data dependencies.

As is also evident from Figure 7, each ICAP cluster was hosted by a single master—a dyadic IBM 3081 on ICAP/1 and a four-processor IBM 3084 on ICAP/2. The greater computing power of the IBM 3081 eliminated the need for a dual-host system on ICAP/1, which had previously employed IBM 43XX systems. As before, ICAP/1 allocated a number of dedicated attached processors specifically for parallel program development and the remainder for production. However, ICAP/2 was used exclusively for production. Again, the IBM 3081 and the 3084 ran under the VM/SP and MVS operating systems, respectively.

An appreciation for the demands imposed by communication can be gathered from the following. The partitioning of a parallel program and its required communication structure is dictated by the operational and data dependencies imbedded within the physics of the application. For example, a single-dimensional fast Fourier transform exhibits a butterfly type of data flow graph. When partitioned among a number of parallel processors, the resulting code requires that the interprocessor connect network mimic this graph well to achieve useful efficiency.[19] Another example could be the solution of time-dependent partial differential equations, often solved

Figure 7 Extended ICAP/1 (VM)





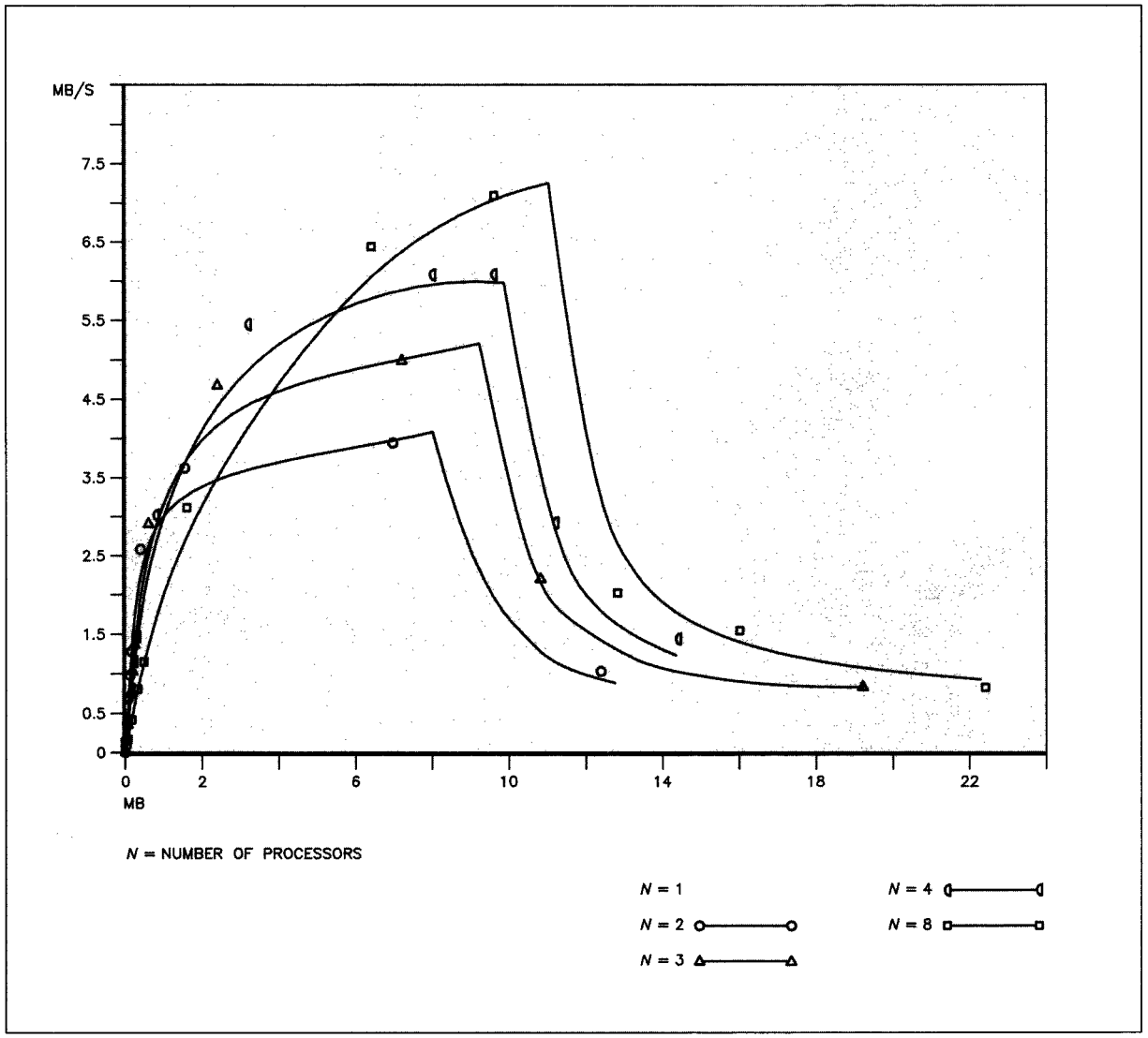
by explicit time-marching schemes. Such problems may be cast in a parallel form by dividing the region of solution equally among different processors (domain decomposition) and requiring synchronization between strictly neighboring processors at the end of every time step.[20] Both of these examples require the ability of some network or shared memory to achieve a variety of fast 1:1 processor intercommunication paths.

Thus, for example, a hypercube multiprocessor system may be effective at performing in parallel a single-dimensional fast Fourier transform because the butterfly network is implicitly imbedded within its structure. Similarly, any multiprocessor configuration that manifests a line, ring, or grid interconnection may be effective for the second example.

On the other hand, an application for which parallel decomposition requires frequent global communication may provide a more demanding test of the efficiency of the interconnection scheme. The requirement for global communication is evident in such applications as molecular dynamics, where one processor is responsible for receiving partial forces from all others (i.e., a receive-from-all operation) and then broadcasting new positions to all slave processors. A broadcast ability is also crucial in many other applications such as celestial mechanics calculations[21] or basic algorithms involving Householder reductions.[22] The latter are typically used in QR factorization for linear least-squares problems or reduction to tridiagonal forms for eigenvalue determination of symmetric systems. Thus, for purposes of illustration, we consider the performance of broadcasting on the ICAP/1 system as a function of communication path and the identity of the broadcaster [i.e., host-to-AP(s) through IBM channels; AP-to-AP(s) by the use of shared memory; and AP-to-AP(s) by the FPS bus]. The conclusions based on these comparisons will hold in an analogous manner for the ICAP/2 cluster.

First, we consider the host-to-AP paths via the IBM channels. Shown in Figure 8 is the measured broadcast bandwidth in MB/S versus the number of MB received in total by the slaves. *Broadcast* bandwidth is defined as the rate at which this quantity of data is transferred. Illustrated are representative cases where the host broadcasts to one, two, three, four, and eight attached processors. The smooth lines through the data points represent a performance model discussed in detail in Reference 23, for which the agreement with experiment was good.

Figure 8 Channel broadcast performance



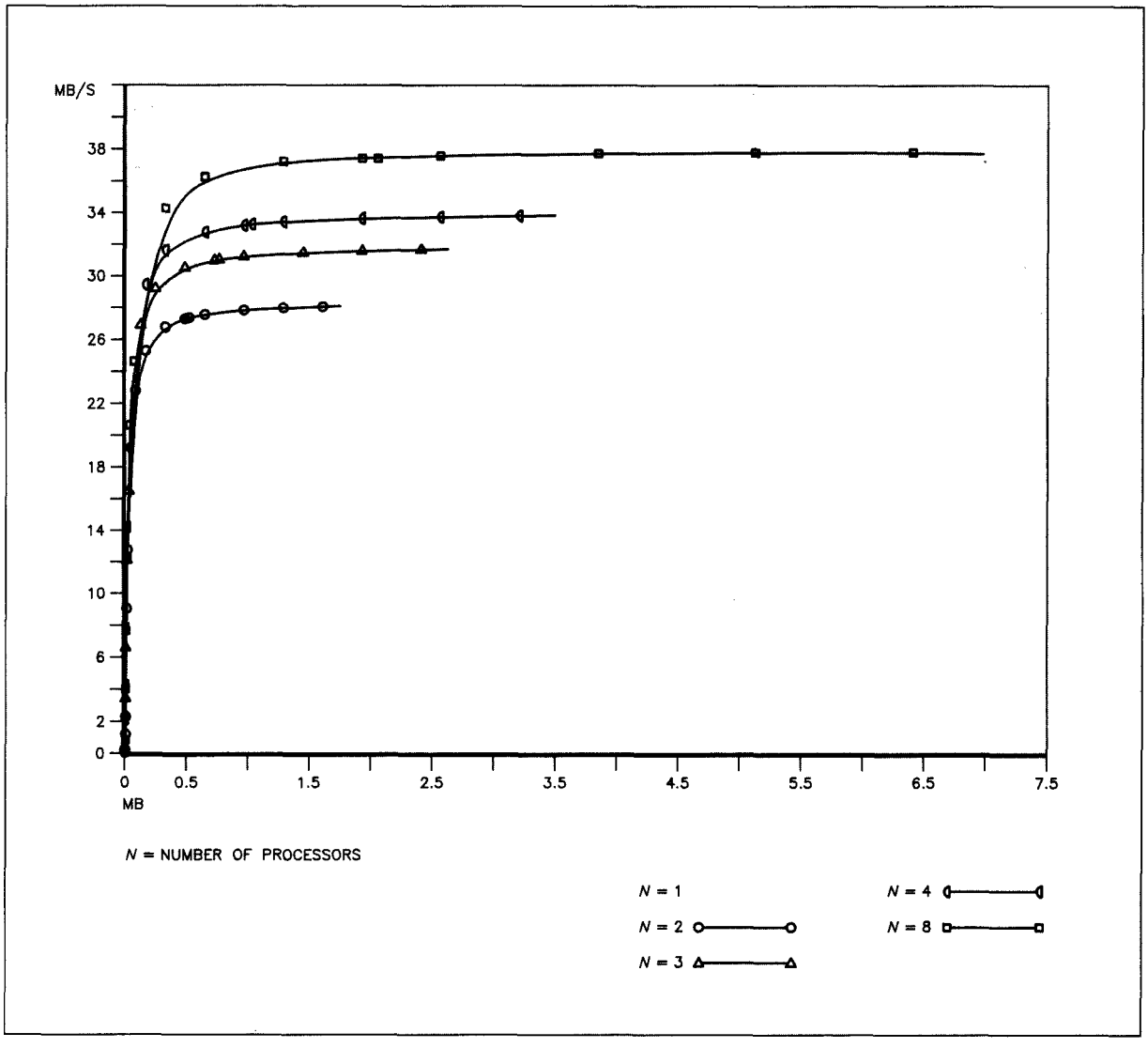
Several observations may be made from a cursory examination of this figure. First, the broadcast bandwidth increases with an increase in number of processors. This is a consequence of employing independent channels for each processor, and thus part of the broadcast operation proceeds in parallel. It is also evident that between approximately 6 and 11 MB of total data sent (in going from 2 to 8 APs), a severe degradation in performance occurs. This behavior was found to be due to paging on the host system. Performance is relatively poor for small transfers. This behavior is attributable to latency effects in initiating the transfers. As data size increases, the transmission rates approach superior asymptotic values prior to the point at which paging degradation occurs.

Next we consider the same operation, but now proceeding by one attached processor, designated as a master, broadcasting to N other attached processors by way of one shared memory. We recall that a shared memory is multiplexed rather than being multiported, and thus it can be accessed by only one processor at a time. Accesses consist of fetching or

storing blocks of contiguous words. Such accesses, once started, cannot be interrupted by another processor. The measured performance for this operation is shown in Figure 9, where the abscissa is the broadcast bandwidth and the ordinate is the total of the MB received. The smooth lines represent a model discussed in detail,[23] for which agreement with experiment was excellent. Indicated are representative cases for broadcasting to 1, 2, 3, 4, and 8 processors.

Several observations may be made regarding this broadcast path. First, compared with channel broadcasting, latency effects, although still present, are of less significance as transfer rates approach their asymptotic values more rapidly for smaller amounts of data. Further, it is clear that bandwidth increases as the number of receivers increases. This is because the number of accesses to bulk memory increases as $N+1$, where N is the number of receiving processors, and the total data transferred increases as N. As time is roughly proportional to $N+1$, we expect as N becomes large that the cumulative rate will approach that at which a single processor can access memory, i.e., nominally 44 MB/S. For a broadcast to one processor, we expect the order of one-half this value, and the observed rates support this expectation.

Figure 9 Shared-memory broadcasting performance



Second, the broadcast operation has been performed by using one memory only. This is a worst-case constraint, as performance may be enhanced significantly by the host broadcasting to several shared memories and letting the recipients receive the data parallel in time from different memories. The present method dictates that only one processor can access the memory at a time, and thus requires, as stated, $N + 1$ memory operations following sequentially in time.

We retain this method for simplicity of comparison with the other communication paths, but it must be borne in mind that distributed shared-memory broadcasting may be considerably faster.

The broadcast bandwidth increases nearly linearly with the number of receiving processors.

Third, we need to consider in more detail the actual implementation of the broadcast operation. The bulk shared memories are passive devices. They have no hardware capability for synchronization. We require, however, that our receiving processors perform their fetch operation only after the broadcaster has completed its store. Thus, we require a synchronization between these operations, which we have effected by a software barrier synchronization that employs dedicated shared-memory addresses for this purpose. This overhead is included in the performance measurements.

Finally, we compare the broadcast performance of the shared-memory route versus that of the host-to-slave-channels path. This is illustrated in Figure 10, where we plot the ratio of times for the slave-channels path over that for the shared memory, as determined from the models, as a function of the logarithm of the total number of bytes received for differing numbers of receiving processors.

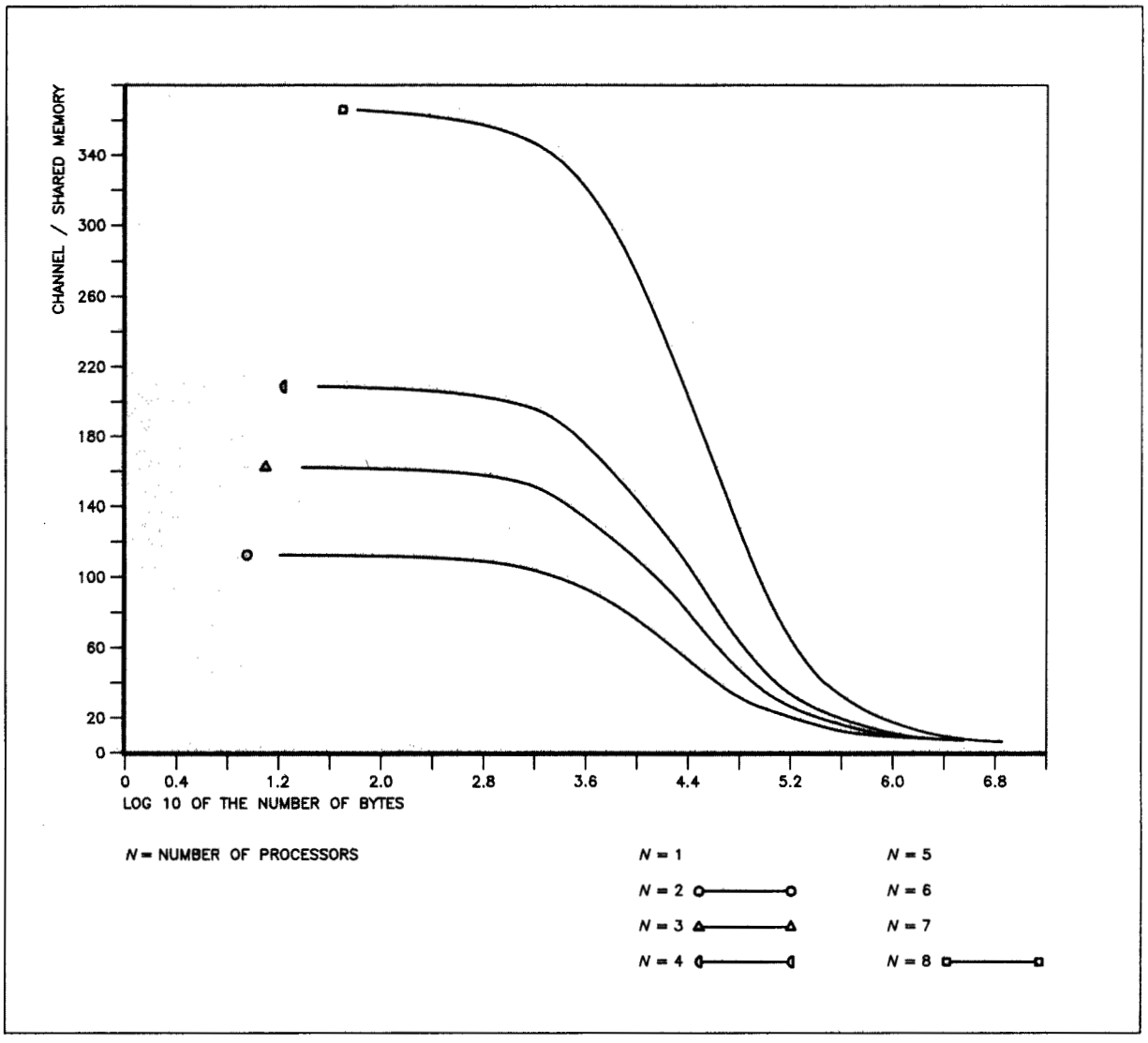
Figure 10 shows that in the asymptotic regions the shared-memory route is roughly ten to five times faster than channel broadcasting, as the number of receivers increases from one to eight. However, in this regime, the order of millions of bytes of data are being transferred. The region that is more important in terms of normal parallel applications deals with transfers of words to kilobytes of data. Here it is seen that the rates favor the shared-memory path by a factor of 300 to 50 over the same range of number of receivers. An examination of the performance models indicates that these superior rates are due almost entirely to significantly smaller latencies in accessing shared memory compared to channel initiation. It is this performance differential that has made the shared memories suitable for much-smaller-grain parallel processing, where the channel communication was previously marginal or inadequate.

Finally, we consider the fast bus. The measured performance of the broadcast bandwidth is plotted in Figure 11 for one to five processors. Again, the lines represent a model discussed in detail in Reference 23, for which agreement with experiment was excellent.

Asymptotically, as expected, the broadcast bandwidth increases nearly linearly with the number of receiving processors. However, the low rates for small packets of data indicate latency effects. Also, packets cannot exceed 32 768 words, and transfers exceeding this limit are sent in separate packets. This is responsible for the zigzag behavior observed in the model, to which the data points fit quite well.

Figures 8, 9, and 11 show that the bus has much better performance than the channels over all data sizes, and performance is somewhat better than the shared-memory path in the asymptotic region where very large transfers are performed. However, in the small-transfer region of words to kilobytes of data, Figure 12 shows that the shared-memory path is roughly an order of magnitude faster than the bus. Here is plotted the ratio of times for broadcasting via the bus over that for the shared memory path (calculated by the models) as a function of the logarithm of the total number of bytes received for differing numbers of receiving processors. Again, the superior rates using the shared memory for small transfers were attributable to its smaller access latency. Overall, the shared-memory path is deemed superior for the larger part of parallel applications that require the broadcast function.

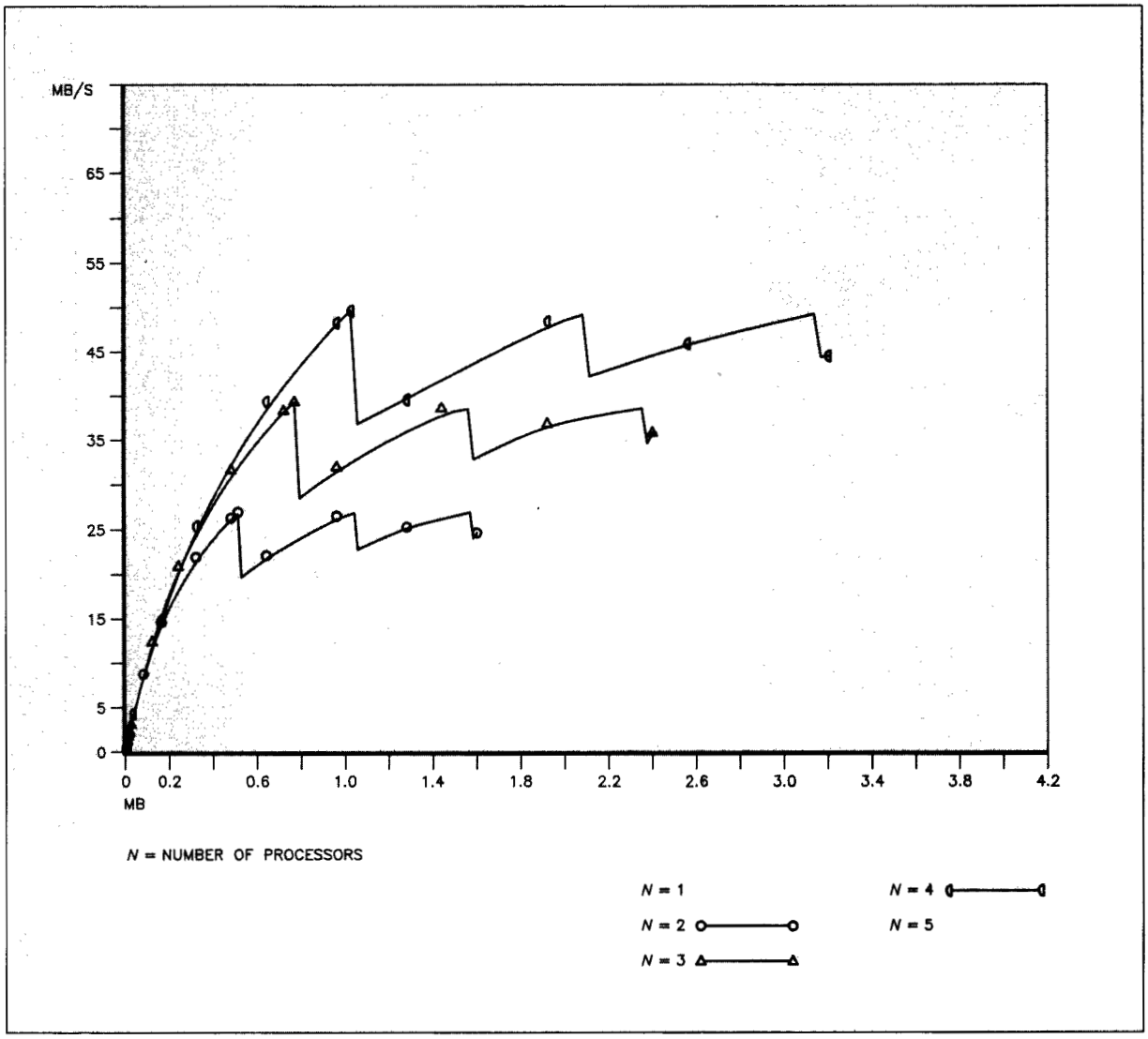
Figure 10 Ratio of broadcast times — channel per shared memory



Although we have singled out the broadcast function for comparison between different communication paths, the conclusions reached hold for other communication structures. They include receive-by-one-processor from all others and any individual processor-to-processor transfers and synchronizations. We have also found that software spin locks of either Peterson's[24] or Lamport's[25] form can be effectively implemented by use of shared memory. This latter function is crucial for applications that require dynamic load balancing through the use of critical, locked paths for task assignments. From the performance results for small data transfers, it may be appreciated that latency factors are of paramount importance in communication for practical parallel applications. That is, asymptotic performance—although important—is a secondary consideration when dealing with limited amounts of interprocessor transfer.

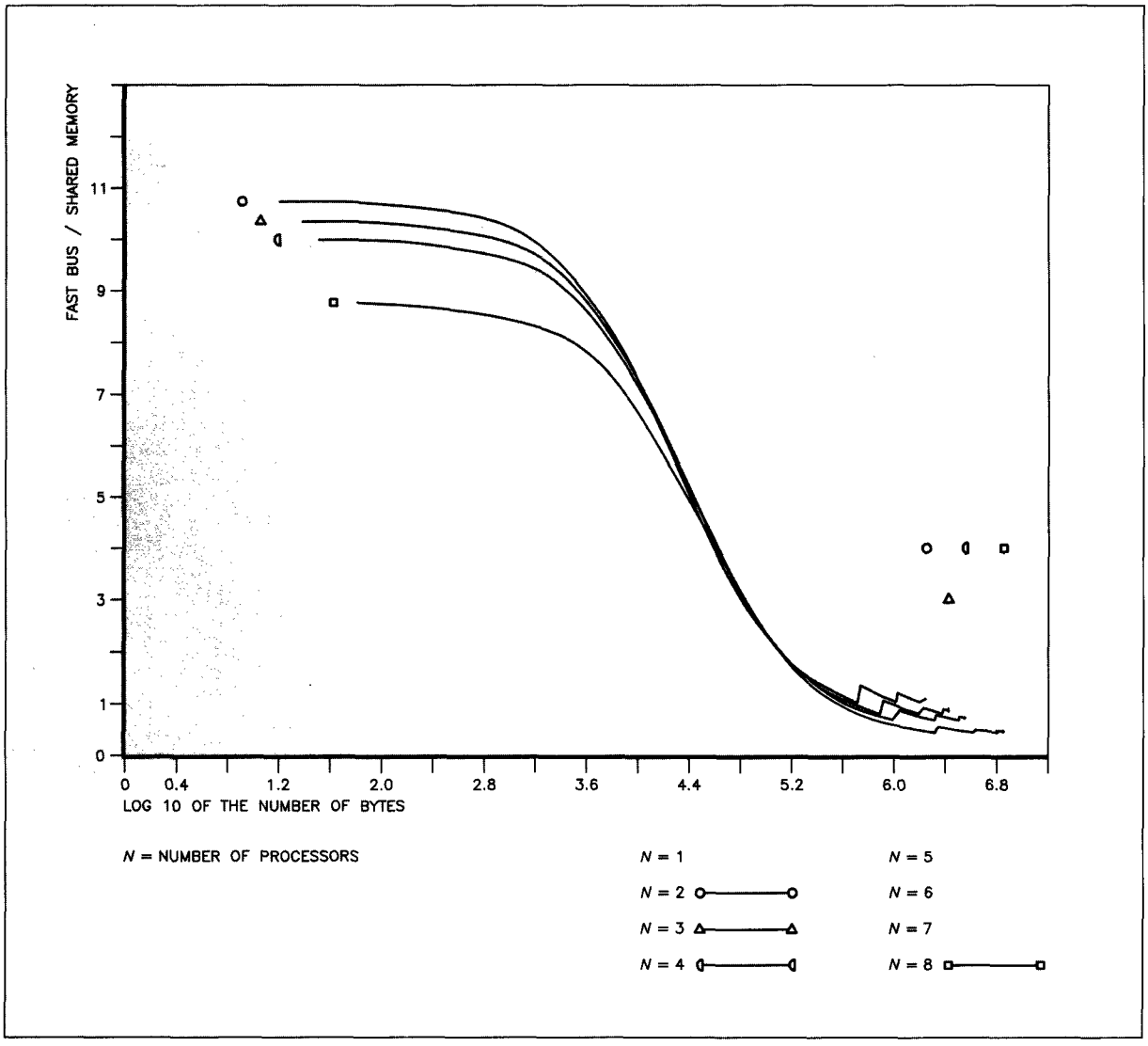
Finally, to bear witness to the effect improved communication plays in executing parallel programs of

Figure 11 Fast bus broadcasting performance



smaller parallel grain size, consider a simple Metropolis Monte Carlo application. Here we simulate the behavior of 512 water molecules, confined in a periodic box and subject to short-range two- and three-body potentials. The portion of the code that is parallelized is the determination of the change in energy when a randomly selected molecule is randomly moved. Each parallel task computes an independent contribution to the energy change and then transmits it to the master task, which then performs the summation over these contributions and accepts or rejects the move. The master then broadcasts the conditionally new coordinates of the molecule for a new cycle on a new randomly chosen molecule. The function of the master may be assumed by the host, in which case communication proceeds via channels, or it may be assumed by one of the attached processors, where communication is performed using either shared memory or the fast bus. In the latter case, the master processor, before assuming that role, also performs one of the parallel tasks.

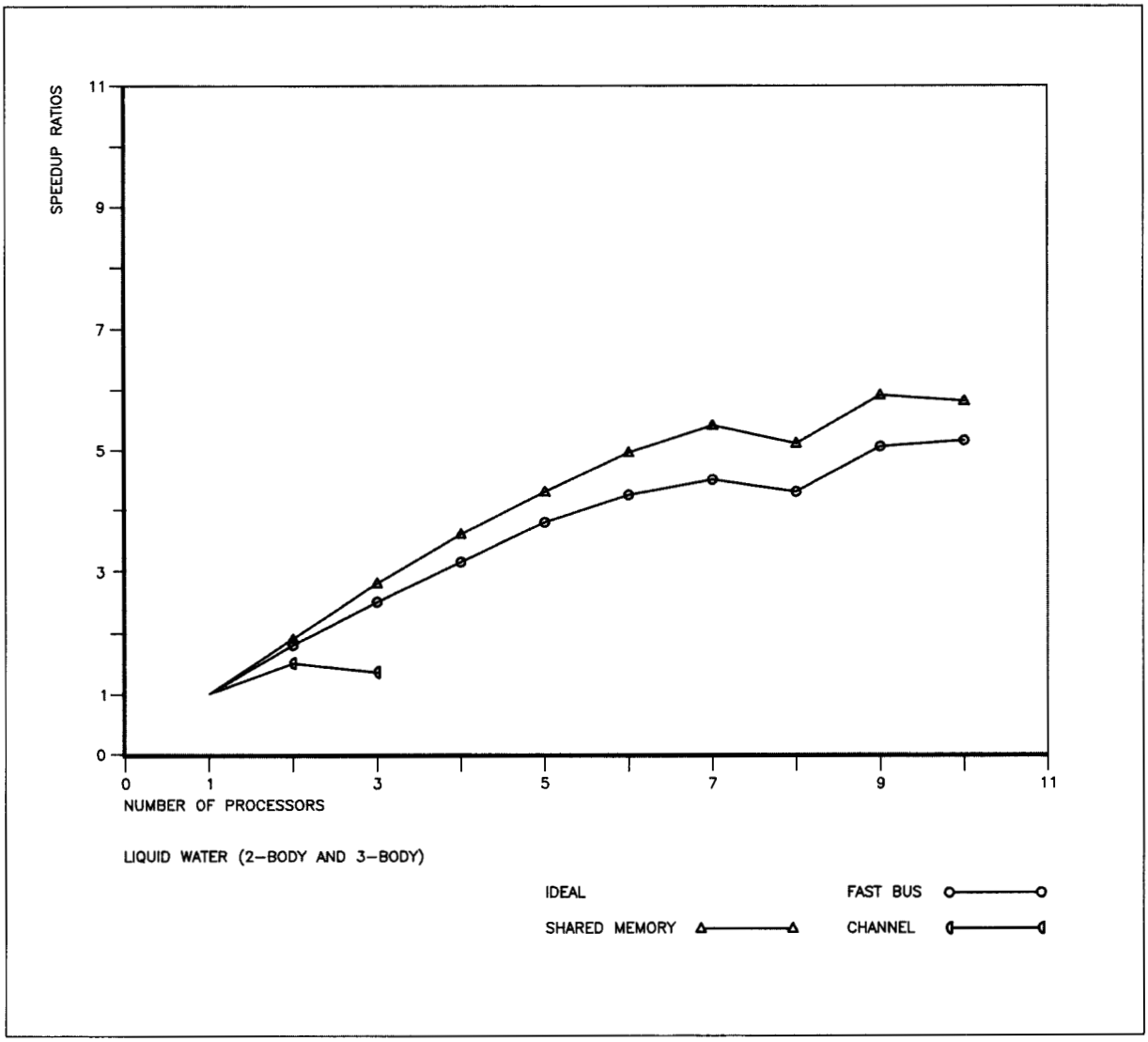
Figure 12 Ratio of broadcast times — fast bus per shared memory



Given that the potential is of short range, the amount of parallel computation is quite small. Further, the quantity of data transmitted back and forth is of the order of tens of words. This description of the program and its expected performance is consistent with the observed speedup curves shown in Figure 13, indicating the superiority of the shared-memory path. Thus, where the channel path (using the faster IBM 3081 as master) gave only marginal improvement for two processors and thereafter actually slowed down, the shared memory permitted reasonable speedup over the entire range of processors. The performance of the bus is better than that of the channel path, but it is inferior to that of the shared memories. This is also consistent with the small data transfers involved.

It is important to point out that none of these communication paths is mutually exclusive. Indeed, transfer to and from the host is of course imperative for program initiation and often optimal for the storage of results. In the latter case, it often occurs

Figure 13 Metropolis Monte Carlo performance



that very large data structures must periodically be saved, and, in this case, the asymptotic capabilities of the channels are wholly adequate. Similarly, the bus can prove to be optimal when very large transfers are required between attached processors, although in reality most of today's applications do not require this ability. As we begin to attack much larger problems, this capability will become more important.

Applications. With these improvements in communication, all of the programs that exhibited marginal to poor performance on the early ICAP systems performed adequately. The following is a partial listing of application areas for which codes have been successfully parallelized on either the initial or the extended systems. References are included where more detailed descriptions of implementation may be found. All of these applications obtained from 60 percent to 95 percent of linear speedup, when partitioned on up to ten processors for problem sizes of interest.

- *Celestial mechanics*—asteroid tracking and collisional probabilities with planets[21]

- *Molecular dynamics*—polymer simulation, thermodynamics of fluids and solutions[26–28]
- *Electronic structure*—properties of molecules, new materials, organic semiconductors[29,30]
- *Fluid dynamics*—turbulence, chaotic behavior[31,32]
- *Micro-hydrodynamics*—basis of Navier–Stokes equations[33,34]
- *Monte Carlo*—Metropolis, quantum Monte Carlo, circuit simulation[14]
- *Circuit analysis*—equivalent capacitance[22]
- *Protein structure*—interferon, membrane mechanisms, DNA studies[14,35]
- *High-energy physics*—data analysis, detector simulation[36]
- *Neutron transport*—reactors
- *Atmospheric studies*—pollution migration[37]
- *Seismic migration*—oil exploration[38]
- *Oceanography*—current flow[39,40]
- *Circuit optimization*—simulated annealing[41]
- *Graphics*—ray tracing[42]
- *Image processing*—parameter estimation, smoothing[43]

The same statement may be made with respect to a large number of parallel algorithms that follow. These are of critical importance in being able to parallelize engineering applications, but they also are of general use in many scientific areas.

- *Linear system solvers*—LU decomposition, Cholesky decomposition, conjugate gradient (preconditioning)[22,44]
- *PDE solution methods*—Alternating Direction Implicit (ADI), MultiColor or Line Successive Over Relaxation (MCSOR and LSOR), multigrid[20,45,46]
- *Eigenvalue solvers*—Jacobi and Householder reductions[22]
- *Factorization*—QR (Givens reductions), QR (Householder reductions), singular value decomposition[22,47,48]
- *Fast Fourier transforms*—mixed radix, Cooley–Tukey[19]
- *Linear programming*—Simplex[49]
- *Statistical database analysis*[50]

The parallel scheduler. As stated previously, an ICAP cluster was essentially a single-user system. That is, there existed no efficient means of dynamically sharing processors among multiple parallel jobs. To make the system more responsive to a general-user environment, we developed a parallel scheduler to perform the sharing. The experience gained in this effort was important in predicting the parallel performance that we could expect with ICAP/3090.

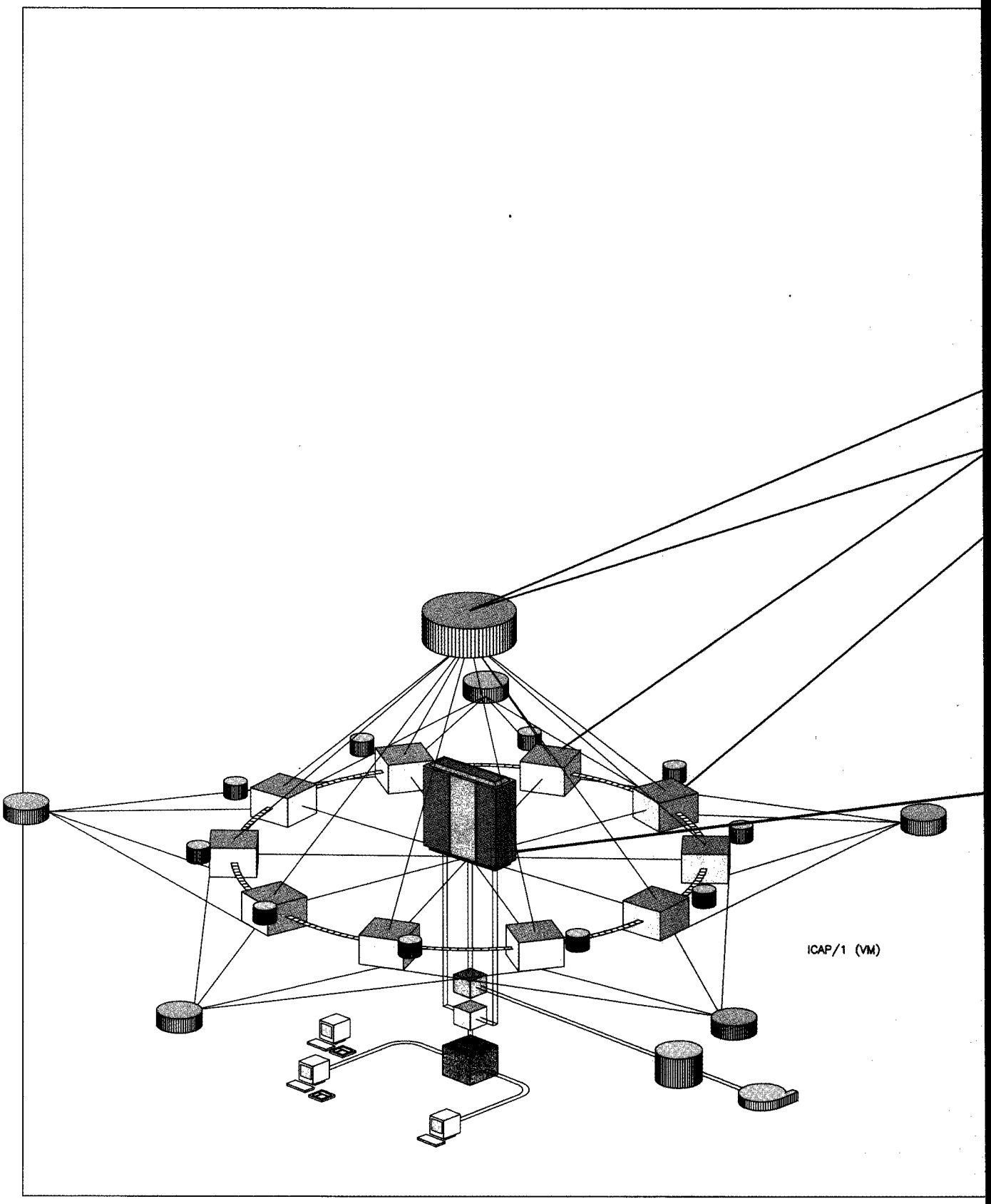
The specification of a parallel job scheduler had to address the following points. First, although context switching between multiple jobs on an FPS processor is permitted by standard FPS operating system software, the time required for state saving is large. The reason is that the memory management system is a real and not a virtual system. Thus, when a context switch occurs, the entire memory allocated to a specific job must be rolled out onto the disk subsystem. For our parallel scheduler, this dictated to a large extent the size of a task's time slice that could be profitably assigned to an attached processor without degrading performance by continually rolling in and out parallel jobs.

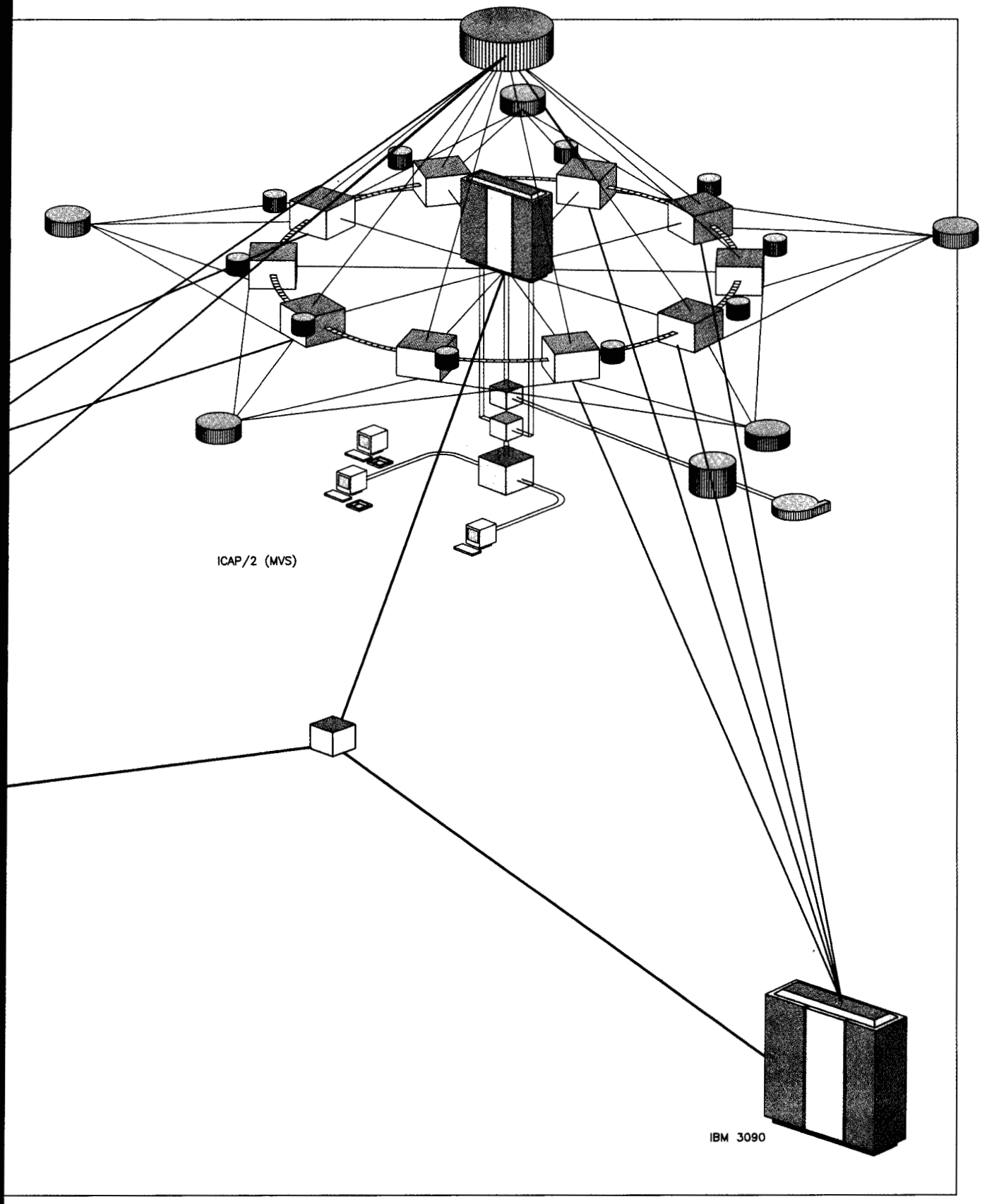
Second, given that we are rolling in and out parallel jobs, it will (in general) be unprofitable to roll out only one or several processors and leave others to continue for a given job. This is because the parallel tasks within a job are typically constrained in time by the need to communicate. Rolling out one task usually results in all the other tasks remaining idle while attempting to communicate with a task that is not there. Thus, an important design criterion for the scheduler was that if a job was designated to be rolled out, all parallel tasks within that job were to be deallocated from their processors at the same time.

With these two constraints, the remainder of the scheduler specification concerned that usually associated with traditional operating systems. We required a priority system that would assign to jobs the preemptive abilities needed to roll out other jobs with lower priority. For our system we decided that, all things being equal, priority would be assigned solely on the user-specified, anticipated execution time. Lower times were assigned higher priority. Thus the scheduler uses a fixed number of priority queues corresponding to certain intervals of user-specified anticipated running time. Penalties were enforced for jobs that exceeded their specified run intervals by placing them in the lowest priority queue. Some fine-tuning was introduced so as not to allow roll-out of lower-priority jobs if they were within a certain fraction of completion of their specified run time. A more detailed description of the scheduler and the scheduling policy may be found in References 10 and 51.

The scheduler resided in a program that ran on a separate virtual machine on the host, through which all parallel jobs were initially assigned resources and thereafter monitored. It had a continuously updated

Figure 14 ICAP/3: coupling of ICAP clusters




ICAP/2 (MVS)
IBM 3090

picture of the demands on the system and performed all accounting information. With the inception of the scheduler, it was found that production and parallel program development jobs could coexist quite well.

Extendability. A natural concern that arises upon proof that a parallel architecture works well across a wide spectrum of applications is *extendability*. How can we make the system more powerful, so that we can address even larger problems? Two (not necessarily mutually exclusive) approaches can be taken. First, we can attempt to replicate and couple these systems together in some efficient manner. Second, we may attempt to replace the nodes or processors with more powerful nodes, contingent upon the resulting system's retaining a high efficiency within the bounds imposed by communication degradation. Both approaches have been explored with the ICAP systems, and the lessons learned have had a strong bearing on the design of ICAP/3090.

Wherever a point-by-point multiplication is required in the traditional algorithm, in the block algorithm this corresponds to a matrix-by-matrix multiplication.

First, an attempt to link the two ICAP clusters was effected by linking the hosts of each system to a super-host, which was an IBM 3090-200 with two processors and vector attachments. Linkage was achieved by channel-to-channel connections. This coupling served a supervisory function for initiating separate clusters. The coupling was made closer by additionally assigning to the 3090 its own subset of attached processors from within each cluster. Later, still closer coupling was achieved by allowing processors from within one cluster to connect to the global shared memory of the other cluster. The peak processing rate of the entire complex was over a gigaflop, with over a gigabyte of memory. A sample configuration, called ICAP/3, is shown in Figure 14.

Several efforts to apply the system to a single problem have met with reasonable success. However, this was achieved at the expense of considerable effort in programming, principally because of the heterogeneity of the system. That is, within the complex there are nodes with large differences in computing power. The requirement of load balancing (i.e., keeping all the nodes busy all of the time) was difficult. An important conclusion was that if extendability is to be achieved by a geometric increase in the system, it is important that all nodes be of equal power. Homogeneity is essential.

We considered the alternative strategy of increasing the computing power of the nodes within a cluster. To a certain extent this issue had already been examined within ICAP/1 and ICAP/2 by replacing FPS-164s with FPS-264s in ICAP/2. We explored this issue further by focusing exclusively on ICAP/1 and incrementing the power of the FPS-164s by adding on arithmetic accelerator boards. More specifically, we incorporated two FPS MAX boards with each processor. The MAX board, designed and marketed by FPS, contains two floating-point multipliers and two floating-point adders that may be operated concurrently with the basic processor. Each board thus adds a potential increase in peak processing power of 22 MFLOPS, and up to 16 boards may be added to each processor. The higher-level software to use these boards was written by FPS and addressed one capability in particular—the ability to perform matrix–matrix multiplication.

While this might seem to be an overly isolated improvement for general performance, it is worth noting that this operation is critical for a broad category of parallel applications. This deals primarily with block algorithms for the solution of important linear-algebraic problems. Block algorithms differ from traditional algorithms in that they operate with submatrix blocks instead of by a datum-by-datum process. Thus, wherever a point-by-point multiplication is required in the traditional algorithm, in the block algorithm this corresponds to a matrix-by-matrix multiplication.

The use of these boards for several parallel-block algorithms was explored on the ICAP/1 system by C. Van Loan and C. Bischof of Cornell University. They studied block versions for symmetric eigenvalue problems, singular value decompositions,[47] and QR factorization.[48] They met with marked improvement in solution times with little additional degradation due to now relatively slower communi-

cation. In all cases, the shared memories were employed. For example, Bischof found the parallel execution of a two-sided block-Jacobi, singular value decomposition performed at a sustained rate of over 160 MFLOPS on large problems with eight processors, each equipped with two MAX boards. This was considerably faster than running the same algorithm without these accelerators.

Concluding remarks

The potential of using parallel processing to solve large-scale problems in science and engineering has been realized with ICAP-type systems. The idea of coupling commercially available processors with simple interconnects has proven successful and can be implemented in a short period of time. This top-down approach has significant advantages compared to inventing new hardware and software.

It is important when constructing a parallel processing system that the applications to be adapted to it are fully understood. There are three factors that degrade parallel performance. The first is the parallel content of the application. Amdahl's law limits the number of concurrent processes that can be effectively applied. The second concerns communication. The speed of communication must be adequate, so that the required communication does not severely degrade performance. This means that the parallel grain size of the problem, its dependence on problem size, and the number of parallel processing elements employed must be fully understood. The third factor concerns load balancing. It is important to have all processors busy all of the time. This requires a careful study of the physics of the application to determine a parallel-task-scheduling policy. The latter can be dynamic or static, but it must make sense in terms of the parallel grain size and the communication capabilities. All three factors must be carefully examined in developing a parallel processing system.

The development of the ICAP/3090 system is a natural outgrowth of our previous experiments. We had proved that a large fraction of important scientific and engineering calculations could be characterized by large- or medium-grain parallelism and could be effectively executed on loosely coupled or shared-memory-based parallel processing systems. Further, the coupling of clusters seems to be an expedient manner in which to extend the power of the system. Overall, the unmatched reliability of the 3090 system combined with its superior scalar, vector, and parallel capabilities makes this an optimal building block for assembling a large-scale parallel processing system.

Cited references and notes

1. M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb, "The Warp computer: Architecture, implementation, and performance," *IEEE Transactions on Computers* **C-36,** No. 12, 1523–1538 (1987).
2. J. A. Fisher, "Very long instruction word architecture: Supercomputing via overlapped execution," *Proceedings of the Second International Conference on Supercomputing* **1,** 353–361 (1987).
3. J. Beetem, M. Denneau, and D. Weingarten, "The GF11 parallel computer," *Special Topics in Super Computing, Vol. 1, Experimental Parallel Computing Architectures*, J. J. Dongarra, Editor, Elsevier Science Publishers, New York (1987).
4. D. M. Nosenchuck, M. G. Littman, and W. Flannery, "Two dimensional nonsteady viscous flow simulation on the Navier Stokes Computer MiniNode," *Journal of Scientific Computation* **1,** No. 1, 53–74 (1986).
5. E. Clementi, G. Corongiu, J. Detrich, S. Chin, and L. Domingo, "Parallelism in computational chemistry: Hydrogen bond study in DNA base pair as an example," *International Journal of Quantum Chemistry Symposium* **18,** 601–618 (1984); IBM Technical Research Report POK-39 (1984); available from Clementi, Logan, or Saarinen.
6. J. H. Detrich, G. Corongiu, and E. Clementi, "Monte Carlo liquid water simulation with four-body interactions included," *Chemical Physics Letters* **112,** 426–430 (1984); IBM Technical Research Reports POK-37 (1984) and KGN-03 (1984); available from Clementi, Logan, or Saarinen.
7. D. H. Gibson, D. W. Rain, and H. F. Walsh, "Engineering and scientific processing on the IBM 3090," *IBM Systems Journal* **25,** No. 1, 36–50 (1986).
8. Y. Singh, G. M. King, and J. W. Anderson, "IBM 3090 performance: A balanced system approach," *IBM Systems Journal* **25,** No. 1, 20–35 (1986).
9. D. L. Meck, *Parallelism in Executing FORTRAN Programs on the 308X: System Considerations and Applications*, IBM Technical Report POK-38 (1984); available from Clementi, Logan, or Saarinen. For another set of FORTRAN-callable communication subroutines to support parallel execution on the IBM 308X under MVS, see IBM Program Offering 5798-DNL, developed by P. R. Martin; the Program Description Operations Manual for this program offering is Order No. SB21-3124; IBM Corporation, available through IBM branch offices.
10. J. Detrich, D. Folsom, and L. Rosenzweig, "ICAP/3090 at IBM Kingston: Evolution of software to support parallel execution," *Proceedings of the 3rd International Conference on Supercomputers* **1,** 99–108 (May 1988).
11. *Parallel FORTRAN Language and Library Reference*, SC23-0431-0, IBM Corporation; available through IBM branch offices.
12. *Virtual Machine/System Product System Programmer's Guide*, Third Edition (August 1983), SC19-6203-2, IBM Corporation; available through IBM branch offices.
13. *MVS/System Product, Version 2, Release 1, General Information*, GC28-1118, IBM Corporation; available through IBM branch offices.
14. *Lecture Notes in Chemistry* **44,** M. Dupuis, Editor, Springer-Verlag, Berlin (1986); *Structure and Dynamics of Nucleic Acids, Proteins and Membranes*, E. Clementi and S. Chin, Editors, Plenum Publishing Company, New York (1986).

15. *ASTAP* is an IBM Installed User Program (IUP), Program No. 5796-PBH.
16. J. M. Ortega and R. G. Voigt, *Solution of Partial Differential Equations on Vector and Parallel Computers*, SIAM Press, Philadelphia (1985).
17. *Shared Bulk Memory System Software Manual, Version 2.0*, Scientific Computing Associates, Yale University, New Haven, CT (1987).
18. *FPSBUS Software Manual, Release G*, Publication No. 860-7313-004A, Floating Point Systems Inc., Beaverton, OR (1986).
19. Z. D. Christidis, V. Sonnad, and D. Logan, *Parallel Implementation of a 2D Fast Fourier Transform on a Loosely Coupled Array of Processors*, IBM Technical Report KGN-68 (1986); may be obtained from Clementi, Logan, or Saarinen.
20. R. Herbin, W. D. Gropp, D. E. Keyes, and V. Sonnad, *A Domain Decomposition Technique on a Loosely Coupled Array of Processors*, IBM Technical Report KGN-124 (1987); may be obtained from Clementi, Logan, or Saarinen.
21. A. Milani, M. Carpino, and D. Logan, *Parallel Computation of Planet Crossing Orbits*, IBM Technical Report KGN-161 (1987); may be obtained from Clementi, Logan, or Saarinen.
22. E. Clementi, D. Logan, and V. Sonnad, "Parallel solution of fundamental algorithms using a loosely coupled array of processors," *Numerical Algorithms for Modern Parallel Computer Architectures*, M. Schultz, Editor, Springer-Verlag, Berlin (1988).
23. D. Logan, J. Saarinen, and E. Clementi, "ICAP/3090: Genesis and evolution of a parallel processing system," *Proceedings of the 3rd International Conference on Supercomputers* **1**, 79–98 (May 1988).
24. G. L. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters* **12**, No. 3, 115–116 (1981).
25. L. Lamport, *A Fast Mutual Exclusion Algorithm*, System Research Report, Digital Equipment Corporation, Systems Research Center, 130 Lytton Ave., Palo Alto, CA, 94301 (1985).
26. M. Bishop, D. Logan, and J. P. J. Michels, "Application of a parallel computer system to polymer calculations," *Theoretica Chimica Acta* **72**, 291–295 (1987).
27. M. W. Evans, G. C. Lie, and E. Clementi, *Molecular Dynamics Computer Simulation of Water From 10 K to 1273 K*, IBM Technical Report KGN-115 (1987); may be obtained from Clementi, Logan, or Saarinen.
28. M. Migliore, G. Corongiu, E. Clementi, and G. C. Lie, *Free Energy for Hydration of Li^+, Na^+, K^+, F^-, and Cl^- With Ab Initio Potentials*, IBM Technical Report KGN-165 (1987); may be obtained from Clementi, Logan, or Saarinen.
29. H. O. Villar, M. Dupuis, J. D. Watts, G. J. B. Hurst, and E. Clementi, *Structure, Vibrational Spectra and IR Intensities of Polyenes From Ab Initio SCF Calculations*, IBM Technical Report KGN-87 (1987); may be obtained from Clementi, Logan, or Saarinen.
30. M. Dupuis, H. O. Villar, and E. Clementi, *Quantum Mechanical Simulations of Polymers for Molecular Electronics and Photonics*, IBM Technical Report KGN-112 (1987); may be obtained from Clementi, Logan, or Saarinen.
31. M. Mansour, A. Garcia, G. C. Lie, and E. Clementi, "Fluctuating hydrodynamics in a dilute gas," *Phys. Review Letters* **58** (9), 874–877 (1987); IBM Technical Report KGN-67 (1986); may be obtained from Clementi, Logan, or Saarinen.
32. L. Hannon, G. C. Lie, E. Clementi, and V. Yakhot, *Fluid-Wall Interactions in Shear Flows: Violation of No-Slip Boundary Conditions*, IBM Technical Report KGN-128 (1987); may be obtained from Clementi, Logan, or Saarinen.
33. D. C. Rapaport and E. Clementi, "Eddy formation in obstructed fluid flow: A molecular dynamics study," *Physical Review Letters* **57** (6), 695–698 (1986); IBM Technical Report KGN-63 (1986); may be obtained from Clementi, Logan, or Saarinen.
34. L. Hannon, G. Lie, and E. Clementi, "Molecular dynamics simulation of flow past a plate," *Journal of Scientific Computation* **1**, No. 2, 145–150 (1986); IBM Technical Report KGN-66 (1986); may be obtained from Clementi, Logan, or Saarinen.
35. K. N. Swamy and E. Clementi, "Hydration structure and the dynamics of B-DNA and Z-DNA," *Biopolymers* **26**, 1901–1927 (1987); IBM Technical Report KGN-94 (1987); may be obtained from Clementi, Logan, or Saarinen.
36. F. Carminati, R. Mount, H. Newman, and H. Pohl, CERN Technical Report L3-313 (1984), EP Division, 1211 Geneva 23, Switzerland.
37. Z. D. Christidis and V. Sonnad, *Parallel Implementation of a Pseudospectral Method on a Loosely Coupled Array of Processors*, IBM Technical Report KGN-143 (1987); may be obtained from Clementi, Logan, or Saarinen.
38. L. Domingo and E. Clementi, *Parallel Computation of Migration of Seismic Data on ICAP*, IBM Technical Report KGN-17 (1985); may be obtained from Clementi, Logan, or Saarinen.
39. A. Capotondi, R. Signell, R. Beardsley, and V. Sonnad, *Tide-Induced Residual Circulation Simulated on a Parallel Computer*, IBM Technical Report KGN-132 (1987); may be obtained from Clementi, Logan, or Saarinen.
40. H. M. Hsu and V. Sonnad, *Parallelization of a Numerical Model for Ocean Circulation*, IBM Technical Report KGN-134 (1987); may be obtained from Clementi, Logan, or Saarinen.
41. Circuit layout optimization on chips has been studied and wiring optimization is also under study.
42. W. Luken, N. Liang, R. Caltabiano, E. Clementi, E. Bacon, J. M. Warren, and W. F. Beausoleil, *Application of Parallel Processing to Molecular Modelling Graphics*, IBM Technical Report KGN-111 (1987); may be obtained from Clementi, Logan, or Saarinen.
43. Parallel application of simulated annealing for image processing has been investigated. The same technique has been applied to Ising model studies of ferromagnetic and antiferromagnetic materials in external fields.
44. G. Brussino and V. Sonnad, "A comparison of preconditioned iterative techniques for sparse, indefinite, unsymmetric systems of linear equations," accepted for publication in *Int. Journal for Numerical Methods in Engineering* (1988); IBM Technical Report KGN-102 (1987); may be obtained from Clementi, Logan, or Saarinen.
45. P. Leca, "Programming loosely coupled multi-FPS system with message passing primitives: Experiment in implementing ADI method on ICAP/1 system," *Proceedings of the Second International Conference on Supercomputing* **2**, 385–391 (1987).
46. R. Herbin, S. Gergi, and V. Sonnad, *Parallel Implementation of a Multigrid Method on the ICAP Supercomputer*, IBM Technical Report KGN-144 (1987); may be obtained from Clementi, Logan, or Saarinen.
47. C. Bischof, *Computing the Singular Value Decomposition on a Distributed System of Vector Processors*, Computer Science Technical Report TR86-798, Cornell University, Ithaca, NY (1987).
48. C. Van Loan, "A block QR factorization scheme for loosely coupled systems of array processors," *Numerical Algorithms for Modern Parallel Computer Architectures*, M. Schultz, Editor, Springer-Verlag, Berlin (1988).

49. Revised simplex-employing explicit inverses have been explored. Karmarkar's algorithm is currently being studied for parallel implementation, necessitating the development of efficient updating in weighted least-squares problems.
50. P. Hopke and L. Kaufman, "An introduction to supercomputers," *Trends in Analytical Chemistry* **6,** No. 1, 1–2 (1987).
51. M. Russo, A. Perez-Ambite, R. Caltabiano, J. Detrich, and D. Folsom, *An Approach to Parallel Scheduling for the ICAP System*, IBM Technical Report KGN-135 (1987); may be obtained from Clementi, Logan, or Saarinen.

Enrico Clementi *IBM Data Systems Division, P.O. Box 100, Kingston, New York 12401.* Dr. Clementi received his Ph.D. in chemistry from the University of Pavia, Italy, in 1954. He did extensive postdoctoral work in experimental and theoretical chemistry with, among others, Nobel Laureates G. Natta at the Polytechnic Institute of Milan (1955) and R. S. Mulliken at the University of Chicago (1960). In 1961 Dr. Clementi joined the IBM Research Division at San Jose, and was responsible for atomic calculations and the publication of atomic tables. From 1967 through 1974 he was manager of a large-scale scientific computation department at IBM San Jose and in 1969 became an IBM Fellow. Dr. Clementi is currently manager of the Scientific and Engineering Computations Department at IBM Kingston, where he has been responsible for research and development in parallel computer architecture, artificial intelligence, and fundamental research in chemistry, biophysics, and fluid dynamics. He has been a recipient of many awards, including nomination as Distinguished Research Professor at Rensselaer Polytechnic Institute, Troy, New York (1986) and the DIRAC golden medal from the World Association of Theoretical Organic Chemists (1987). He has authored and coauthored over 350 papers and is among the top 300 Contemporary Scientists most cited from 1965–1978, according to E. Garfield in *Current Contents* **9,** 5 (1982). Dr. Clementi is a member of many professional societies and a Fellow of the American Physical Society (1984).

Douglas R. Logan *IBM Data Systems Division, P.O. Box 100, Kingston, New York 12401.* Dr. Logan received his B.Sc. degree in chemistry from McGill University, Montreal, Quebec, in 1969 and his Ph.D. in nuclear chemistry from Columbia University, New York, in 1975. Following postdoctoral work in fundamental nuclear reaction research at Carnegie Mellon University, he joined the Advanced Computer Architecture Laboratory at the Lawrence Berkeley Laboratory, Berkeley, California, where he worked as head of applications development for parallel processing on the MIDAS parallel computer. In 1985 Dr. Logan joined the Scientific and Engineering Computations Department at IBM Kingston, where he has worked primarily on parallel algorithm development and performance analysis of parallel architectures. He is an author and coauthor of over 60 papers in nuclear and computer science and was co-winner of the award for best presentation at the 1983 and 1984 International Conferences of Parallel Processing.

Jukka P. Saarinen *IBM Data Systems Division, P.O. Box 100, Kingston, New York 12401.* Mr. Saarinen is currently completing his Ph.D. in computer science from the Tampere University of Technology, Finland, while he is engaged in research on parallel architectures and their use in numerically intensive engineering applications at the Scientific and Engineering Computations Department at IBM Kingston. His research interests include digital signal processing, pattern recognition, image processing and Monte Carlo simulations. Mr. Saarinen is a member of the Electrical Engineering Society of Finland, the Finnish Technical Society, the Pattern Recognition Society of Finland, and the Institute of Electrical and Electronics Engineers.

Reprint Order No. G321-5339.