The NYU Ultracomputer -- Designing a MIMD, Shared-Memory Parallel Machine (Extended Abstract)

Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir

Courant Institute of Mathematical Sciences, NYU
251 Mercer St., New York, NY 10012

Abstract

We present the design for the NYU Ultracomputer, a shared-memory MIMD parallel machine composed of thousands of autonomous processing elements. This machine uses an enhanced message switching network with the geometry of an Omega-network to approximate the ideal behavior of Schwartz's paracomputer model of computation and to implement efficiently the important fetch-and-add synchronization primitive. We outline the hardware that would be required to build a 4096 processor system using 1990's technology. We also discuss system software issues, and present analytic studies of the network performance. Finally, we include a sample of our effort to implement and simulate parallel variants of important scientific programs.

1.0 INTRODUCTION

Within a few years advanced VLSI (very large scale integration) technology will produce a fast single-chip processor including high-speed floating-point arithmetic. This leads one to contemplate the level of computing power that would be attained if thousands of such processors cooperated effectively on the solution of large-scale computational problems.

The NYU "Ultracomputer" group has been studying how such ensembles can be constructed for effective use and has produced a tentative design that includes some novel hardware and software components. The design may be broadly classified as a general purpose MIMD machine accessing a central shared memory via a message switching network with the geometry of an Omega-network. (For related designs see Burroughs [79], Siegel et al. [81], Smith [78], Sullivan et al. [77], and Swan et al. [77].)

---

* Current address University of Illinois, Urbana.

** Current address University of Toronto.

This work was supported in part by the National Science Foundation and the Applied Mathematical Sciences Program of the Department of Energy, grant numbers NSF-MCS79-07804 and DE-AC02-76ER03077 respectively.

The major thrust of this report is to outline and justify, in some detail, the proposed hardware and present the analytic and simulation results upon which parts of the design are based. We also discuss system software issues and describe some of our ongoing efforts to produce parallel versions of important scientific programs (but the reader should see Gottlieb et al. [81] and Kalos [81] respectively for a more detailed treatment of these last two topics). Section 2 of the present report reviews the idealized computation model upon which our design is based; section 3 presents the machine design; section 4 analyzes network performance; section 5 highlights a parallel scientific program; and section 6 summarizes our results.

2.0 MACHINE MODEL

In this section we first review the paracomputer model, upon which our machine design is based, and the fetch-and-add operation, which we use for interprocessor synchronization. After illustrating the power of this model, we examine alternates and justify our selection. Although the paracomputer model to be described is not physically realizable, we shall see in section 3 that close approximations can be built.

2.1 Paracomputers

An idealized parallel processor, dubbed a "paracomputer" by Schwartz [80] and classified as a WRAM by Borodin and Hopcroft [81], consists of autonomous processing elements (PEs) sharing a central memory. The model permits every PE to read or write a shared memory cell in one cycle. In particular, simultaneous reads and writes directed at the same memory cell are effected in a single cycle.

We augment the paracomputer model with the "fetch-and-add" operation (described below) and make precise the effect of simultaneous access to the shared memory. To accomplish the latter we define the serialization principle: The effect of simultaneous actions by the PEs is as if the actions occurred in some (unspecified) serial order. For example, consider the effect of one load and two stores simultaneously directed at the same memory cell. The cell will come to contain some one of the quantities written into it. The load will return either the original value or one of the stored values, possibly different from the value the cell comes to contain. Note that simultaneous memory updates are in fact accomplished in one cycle; the serialization principle speaks only of the effect of simultaneous actions and not of their implementation.

We stress that paracomputers must be regarded as idealized computational models since physical limitations, such as restricted fan-in, prevent their realization. In the next section we review the technique whereby a connection network may be used to construct a parallel processor closely approximating our enhanced paracomputer.

2.2 The Fetch-And-Add Operation

We now introduce a simple yet very effective interprocessor synchronization operation, called fetch-and-add, which permits highly concurrent execution of operating system primitives and application programs. The format of this operation is F&A(V,e), where V is an integer variable and e is an integer expression. This indivisible operation is defined to return the (old) value of V and to replace V by the sum V+e. Moreover, fetch-and-add must satisfy the serialization principle stated above: If V is a shared variable and many fetch-and-add operations simultaneously address V, the effect of these operations is exactly what it would be if they occurred in some (unspecified) serial order, i.e. V is modified by the appropriate total increment and each operation yields the intermediate value of V corresponding to its position in this order. The following example illustrates the semantics of fetch-and-add: Assuming V is a shared variable, if PEi executes

```
          ANSi <-- F&A(V,ei)  ,
```

and if PEj simultaneously executes

```
          ANSj <-- F&A(V,ej)  ,
```

and if V is not simultaneously updated by yet another processor, then either

```
     ANSi <-- V        or     ANSi <-- V+ej
     ANSj <-- V+ei            ANSj <-- V
```

and, in either case, the value of V becomes V+ei+ej.

For another example consider several PEs concurrently applying fetch-and-add, with an increment of 1, to a shared array index. Each PE obtains an index to a distinct array element (although one does not know beforehand which element will be assigned to which PE). Furthermore, the shared index receives the appropriate total increment.

Section 3 presents a hardware design that realizes fetch-and-add without significantly increasing the time required to access shared memory and that realizes simultaneous fetch-and-adds updating the same variable in a particularly efficient manner.

2.3 The Power of Fetch-And-Add

Since in a parallel processor the relative cost of serial bottlenecks rises with the number of PEs, users of future ultra-large-scale machines will be anxious to avoid the use of critical (and hence necessarily serial) code sections, even if these sections are small enough to be entirely acceptable in current practice.

If the fetch-and-add operation is available, we can perform many important algorithms in a completely parallel manner, i.e. without using any critical sections. For example Gottlieb et al. [81]* presents a completely parallel solution to the readers-writers problem** and a highly concurrent queue management technique that can be used to implement a totally decentralized operating system scheduler. We are unaware of any other completely parallel solutions to these problems. To illustrate the nonserial behavior of these algorithms, we note that given a single queue that is neither empty nor full, the concurrent execution of thousands of inserts and thousands of deletes can all be accomplished in the time required for just one such operation. Other highly parallel fetch-and-add-based algorithms appear in Kalos [81], Kruskal [81], and Rudolph [82].

2.4 Generalizing Fetch-And-Add

One can define a more general fetch-and-phi operation that fetches the value in V and replaces it with phi(V,e). Of course defining phi(a,b)=a+b gives fetch-and-add. If phi is both associative and commutative, the final value in V after the completion of concurrent fetch-and-phi's is independent of the serialization order chosen.

We now show that two important coordination primitives, swap and test-and-set, may also be obtained as special cases of fetch-and-phi. (It must be noted, however, that the fetch-and-add operation has proved to be a sufficient coordination primitive for all the highly concurrent algorithms developed to date.) We use the brackets { and } to group statements that must be executed indivisibly and define test-and-set to be a value-returning procedure operating on a shared Boolean variable:

```
     TestAndSet(V)
        { Temp <-- V
          V <-- TRUE }
        RETURN Temp
```

* As explained in Gottlieb and Kruskal [81], the replace-add primitive defined in Gottlieb et al. [81] and used in several of our earlier reports is essentially equivalent to the fetch-and-add primitive used in the present paper.

** Since writers are inherently serial, the solution cannot strictly speaking be considered completely parallel. However, the only critical section used is required by the problem specification. In particular, during periods when no writers are active, no serial code is executed.

The swap operation is defined as exchanging the values of a local variable L (which specifies a processor register or stack location) and a variable V stored in central memory

```
     Swap(L,V)
        { Temp <-- L
          L <-- V
          V <-- Temp }
```

It is easy to see that

TestAndSet(V) is equivalent to FetchOR(V,TRUE).

Similarly, a swap operation can be effected by using the projection operator pi2, where pi2(a,b) = b; i.e.

Swap(L,V) is equivalent to L <-- FetchPi2(V,L).

We conclude this discussion of fetch-and-phi by showing that this operation may be used as the sole primitive for accessing central memory. Specifically, we show how to obtain the familiar load and store operations as degenerate cases of fetch-and-phi. To load the local variable L from a variable V stored in central memory one simply executes

L <-- FetchPi1(V,*)

where Pi1(a,b)=a and the value of * is immaterial (and thus need not be transmitted). Similarly, to store the value of L into V one executes

* <-- FetchPi2(V,L) where the * indicates that the value returned is not used (and thus again need not be transmitted).

2.5 Alternate Machine Models

In this subsection we discuss several other heavily researched models of parallel processors and explain our choice of a large-scale MIMD shared memory machine.

One line of study pioneered by H. T. Kung (see e.g. Kung [80]), focuses on the great economic and speed advantages obtainable by designing parallel algorithms that conform well to the restrictions imposed by VLSI technology, in particular algorithms and architectures that lay out well in two dimensions. These "systolic" processor designs are already having a significant impact on signal processing, an impact that will doubtless increase dramatically over the next several years. However, for computations having complex control and data flow, the systolic architecture is less well suited. We do expect that VLSI systolic systems will be used for those subcomponents of our machine having regular control and data flow; the design of one such component, an enhanced systolic queue, is presented in section 3.5.

The current generation of supercomputers may be roughly classified as SIMD shared memory machines by considering their vector pipelines to be multiple processors each executing the same instruction (cf. Stone [80]). Effective use of such machines is only attained by algorithms consisting primarily of vector operations. Although it is far from trivial to "vectorize" algorithms, such a program has been successfully undertaken at many supercomputer sites. Once again, however, some problems (especially those with many data dependent decisions) appear to resist effective vectorization. Rodrigue, Giroux, and Pratt [80] of Lawrence Livermore National Laboratory write:

> Vector and array processors were designed with the idea of solving fluid-type problems efficiently. In general these machines do not lend themselves well to particle tracking calculations. For a scientific laboratory such as LLNL, the computer should be able to handle both forms of calculation, but it remains to be seen whether this goal will ever be achieved.

This goal is achieved by rejecting SIMD machines in favor of the MIMD paracomputer model, which our simulation studies have shown to be effective for both fluid-type (Rushfield [81]) and particle tracking calculations (Kalos et al. [81]).

Yet a third alternative model, specifically architectures derived from very general abstract "dataflow" models of parallel computation, have been pursued by other researchers (see the February 1982 special issue of Computer and the references contained therein). Recent work in this area has stressed the advantages of a purely applicative, side-effect-free programming language for the description of parallel computation. Although such dataflow machines have been discussed for several years, no completely satisfactory physical design has yet emerged. Without commenting on the relative merits of applicative programming, we note that Gottlieb and Schwartz [81] show how a dataflow language may be executed with maximal parallelism on our machine.

The final model we consider is a message passing alternative to shared memory. Except for very small systems, it is not possible to have every PE directly connected to every other PE. Thus it may be necessary to route messages via intermediate PEs. In the original ultracomputer design of Schwartz [80] the programmer specified the routing explicitly. By tailoring algorithms to the particular interconnection geometry, one can obtain very high performance. However, we found such a machine to be significantly more difficult to program than one in which the entire memory is available to each PE (see Schwartz [80], Gottlieb [80a, 80c], Gottlieb and Kruskal [80]). If the geometry is hidden from the programmer by having the individual PEs perform the necessary routing, a more loosely coupled machine results. In recent years such machines have been much studied for distributed computing applications. Although message passing architectures are indeed quite attractive for distributed computing, we believe that for the applications we have emphasized, thousands of processors cooperating to solve a single large-scale scientific problem, the more tightly coupled model featuring high speed concurrent access to shared memory is more effective.

3.0 MACHINE DESIGN

In this section we sketch the design of the NYU Ultracomputer, a machine that appears to the user as a paracomputer, and we justify our design decisions. As indicated above, no machine can provide the single-cycle access to shared memory postulated in the paracomputer model; our design approximates a paracomputer by using a message switching network with the geometry of the Omega-network of Lawrie* to connect N = 2↑D autonomous PEs to a central shared memory composed of N memory modules (MMs). Thus, the direct single cycle access to shared memory characteristic of paracomputers is replaced by an indirect access via a multicycle connection network. Each PE is attached to the network via a processor network interface (PNI) and each MM is attached via a memory network interface (MNI). Figure 1 gives a block diagram of the machine.

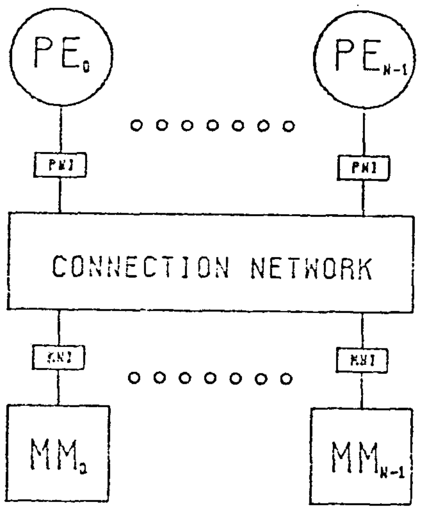

Figure 1. Block Diagram

After reviewing routing in the network, we show that an analogous network composed of enhanced switches provides efficient support for concurrent fetch-and-add operations. We then examine our choice of network and local memory implementation. To conclude this section we present a detailed design for the switches and describe the PEs, MMs, and network interfaces. As will be shown both the PEs and MMs are relatively standard components; the novelty of the design lies in the network and in particular in the constituent switches and interfaces.

3.1 Network Design

For machines with thousands of PEs the communication network is likely to be the dominant component with respect to both cost and performance. The design to be presented achieves the following objectives.

1. Bandwidth linear in N, the number of PEs.

2. Latency, i.e. memory access time, logarithmic in N.

3. Only O(N log N) identical components.

4. Routing decisions local to each switch; thus routing is not a serial bottleneck and is efficient for short messages.

5. Concurrent access by multiple PEs to the same memory cell suffers no performance penalty; thus interprocessor coordination is not serialized.

We are unaware of any significantly different design that also attains these goals.

3.1.1 Routing in an Omega-Network

3.1.1 Routing in an Omega-Network - The manner in which an Omega-network can be used to implement memory loads and stores is well known and is based on the existence of a (unique) path connecting each PE-MM pair. To describe the routing algorithm we use the notation in Figure 2: both the PEs and the MMs are numbered using D-bit identifiers whose values range from 0 to N-1; the binary representation of each identifier x is denoted xD...x1; upper ports on switches are numbered 0 and lower ports 1; messages from PEs to MMs traverse the switches from left to right; and returning messages traverse the switches from right to left.

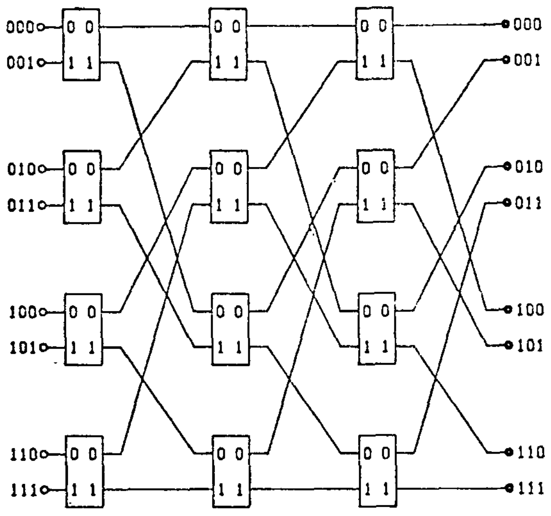

Figure 2. Omega-network (N=8)

* Note that this network has the same topology as a rectangular SW banyan network (see Goke and Lipovsky).

A message is transmitted from PE(pD...p1) to MM(mD...m1) by using output port mj when leaving the stage j switch. Similarly, to travel from MM(mD...m1) to PE(pD...p1) a message uses output port pj at a stage j switch.

The routing algorithm just presented generalizes immediately to a D-stage network composed of k-input-k-output switches (instead of the 2x2 switches used above) connecting k↑D PEs to k↑D MMs: The ports of a switch are numbered 0 to k-1 and the identifiers are written in base k. Although the remainder of this section deals exclusively with 2x2 switches, all the results generalize to larger switches, which are considered in section 4.

3.1.2 Omega-Network Enhancements - To prevent the network from becoming a bottleneck for machines comprising large numbers of PEs, an important design goal has been to attain a bandwidth proportional to the number of PEs. This has been achieved by a combination of three factors (see section 4 for an analysis of network bandwidth):

1. The network is pipelined, i.e. the delay between messages equals the switch cycle time not the network transit time. (Since the latter grows logarithmically, nonpipelined networks can have bandwidth at most O(N/log N).)

2. The network is message switched, i.e. the switch settings are not maintained while a reply is awaited. (The alternative, circuit switching, is incompatible with pipelining.)

3. A queue is associated with each switch to enable concurrent processing of requests for the same port. (The alternative adopted by Burroughs [79] of killing one of the two conflicting requests also limits bandwidth to O(N/log N), see Kruskal and Snir.)

Since we propose using a message switching network, it may appear that both the destination and return addresses must be transmitted with each message. We need, however, transmit only one D bit address, an amalgam of the origin and destination: When a message first enters the network, its origin is determined by the input port, so only the destination address is needed. Switches at the j-th stage route messages based on memory address bit mj and then replace this bit with the PE number bit pj, which equals the number of the input port on which the message arrived. Thus, when the message reaches its destination, the return address is available.

When concurrent loads and stores are directed at the same memory location and meet at a switch, they can be combined without introducing any delay by using the following procedure (see Klappholtz [81], Sullivan and Cohen [79], and Gottlieb et al. [81])

1. Load-Load: Forward one of the two (identical) loads and satisfy each by returning the value obtained from memory.

2. Load-Store: Forward the store and return its value to satisfy the load.

3. Store-Store: Forward either store and ignore the other.

Combining requests reduces communication traffic and thus decreases the lengths of the queues mentioned above, leading to lower network latency (i.e. reduced memory access time). Since combined requests can themselves be combined, the network satisfies the key property that any number of concurrent memory references to the same location can be satisfied in the time required for just one central memory access. It is this property, when extended to include fetch-and-add operations as indicated below, that permits the bottleneck-free implementation of many coordination protocols.

3.1.3 Implementing Fetch-And-Add - By including adders in the MNI's, the fetch-and-add operation can be easily implemented: When F&A(X,e) is transmitted through the network and reaches the MNI associated with the MM containing X, the value of X and the transmitted e are brought to the MNI adder, the sum is stored in X, and the old value of X is returned through the network to the requesting PE. Since fetch-and-add is our sole synchronization primitive (and is also a key ingredient in many algorithms), concurrent fetch-and-add operations will often be directed at the same location. Thus, as indicated above, it is crucial in a design supporting large numbers of processors not to serialize this activity.

Enhanced switches permit the network to combine fetch-and-adds with the same efficiency as it combines loads and stores: When two fetch-and-adds referencing the same shared variable, say F&A(X,e) and F&A(X,f), meet at a switch, the switch forms the sum e+f, transmits the combined request F&A(X,e+f), and stores the value e in its local memory (see Figure 3). When the value Y is returned to the switch in response to F&A(X,e+f), the switch transmits Y to satisfy the original request F&A(X,e) and transmits Y+e to satisfy the original request F&A(X,f).
Assuming that the combined request was not further combined with yet another request, we would have Y = X; thus the values returned by the switch are X and X+e, thereby effecting the serialization order "F&A(X,e) followed immediately by F&A(X,f)". The memory location X is also properly incremented, becoming X+e+f. If other fetch-and-add operations updating X are encountered, the combined requests are themselves combined, and the associativity of

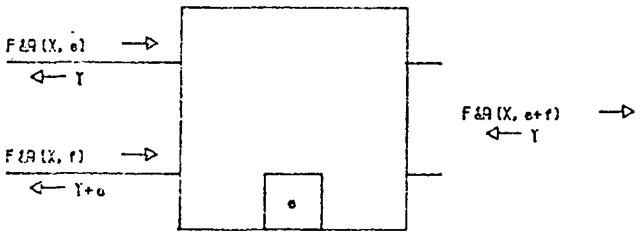

Figure 3. Combining Fetch-And-Adds

addition guarantees that the procedure gives a result consistent with the serialization principle.

Although the preceding description assumed that the requests to be combined arrive at a switch simultaneously, the actual design can also merge an incoming request with requests already queued for output to the next stage (see section 3.4).

To combine a fetch-and-add operation with another reference to the same memory location we proceed as follows:

1. FetchAdd-FetchAdd. As described above, a combined request is transmitted and the result is used to satisfy both fetch-and-adds.

2. FetchAdd-Load. Treat Load(X) as FetchAdd(X,0).

3. FetchAdd(X,e)-Store(X,f). Transmit Store(e+f) and satisfy the fetch-and-add by returning f.

Finally, we note that a straightforward generalization of the above design yields a network implementing the fetch-and-phi primitive for any associative operator phi.

3.1.4 Other Considerations - We now turn our attention to other issues concerning the proposed network design.

Since the introduction of queues in each switch leads to stochastic delays and the network is pipelined, it is possible for memory references from a given PE to distinct MMs to be satisfied in an order different from the order in which they were issued. This reordering can violate the serialization principle specified in our model. A simple-minded solution to this problem is not to pipeline requests to read-write shared variables; however, this approach is overly conservative since most such request can be safely pipelined.

Since the analyses thus far obtained require the introduction of simplifying assumptions (see section 4), and we are unable to perform faithful simulations of full 4096 PE networks, we cannot confidently predict the expected network latency. Our preliminary analyses and partial simulations have yielded encouraging results.

A potential serial bottleneck is the memory module itself. If every PE simultaneously requests a distinct word from the same MM, these N requests are serviced one at a time. However, introducing a hashing function when translating the virtual address to a physical address, assures that this unfavorable situation occurs with probability approaching zero as N increases.

The hardware complexity due to the decision to adopt a queued message switching network introduces significant processing at each stage. Although the internal cycle time of the switches may be important for today's technology, we expect that by the end of the decade any on-chip delay will be dominated by the chip-to-chip transmission delays. (Since the switch bandwidth will be pin limited, the added internal complexity will not increase the component count.)

3.2 Local Memory

The negative impact of the large network latency can be partially mitigated by providing each PE with a local memory in which private variables reside and into which read-only shared data (in particular, program text) may be copied. Storing shared read-write data in the local memory of multiple PEs must, in general, be prohibited: The resulting memory incoherence would otherwise lead to violations of the serialization principle. We shall show in section 3.4 that in certain special cases, this restriction may be relaxed.

One common design for parallel machines is to implement a separately addressable local memory at each PE, imposing upon compilers and loaders the onus of managing the two level store. The alternative approach, which we intend to implement, is the one conventionally used on uniprocessors: The local memory is implemented as a cache. Experience with uniprocessor systems shows that a large cache can capture up to 95% of the references to cacheable variables, effectively shifting the burden of managing a two level store from the software to the hardware (see Kaplan and Winder [73]).

3.3 The Switches

We now detail an individual network switch, which is essentially a 2x2 bidirectional routing device transmitting a message from its input ports to the appropriate output port on the opposite side. The PE side sends and receives messages to and from the PEs via input ports, called FromPEi, where i=0,1, and output ports, called ToPEi. Similarly, the MM side communicates with the MMs via ports FromMMi and ToMMi. (Note that in our figures the To and From ports are coalesced into bidirectional ports.)

As indicated above, we associate a queue with each output port. The head entry is transmitted when the switch at the adjacent stage is ready to receive it (the message might be delayed if the queue this message is due to enter is already full).

To describe the process whereby requests are combined in a switch, we view a request as consisting of several components: function indicator (i.e. load, store, or fetch-and-add), address, and data. The address itself consists of the amalgamation of part of the PE number and part of the MM number, and the internal address within the specified MM. For ease of exposition, we consider only combining homogeneous requests (i.e. requests with like function fields); it is not hard to extend the design to permit combining heterogeneous requests. For each request, R-new, that enters a ToMM queue*, we search the requests already in this queue using as key the function, MM number, and internal address from R-new.** If no request matches R-new, then no combining is possible and R-new simply remains the tail entry of the output queue. Otherwise, let R-old denote the message in the ToMM queue that matches R-new. Then, to effect the serialization R-old followed immediately by R-new, the switch performs the following actions: The addresses of R-new and R-old are placed into a Wait Buffer (to await the return of R-old from memory) and R-new is deleted from the ToMM queue. If the request is a store then the datum of R-old (in the toMM queue) is replaced by the datum of R-new. If the request is a fetch-and-add then the datum of R-old is replaced by the sum of the two data. In addition, for fetch-and-adds, the datum of R-old is sent to the Wait Buffer. Thus, each entry sent to the wait buffer consists of the address of R-old (the entry key); the address of R-new; and, in the case of a combined fetch-and-add, a datum. (Note that stores and fetch-and-adds can both be implemented by using an ALU that receives the data of R-old and R-new and returns either the sum of the two numbers or just R-new.)

Before presenting the actions that occur when a request returns to a switch from a MM, we make two remarks. First, we would use two Wait Buffers (one associated with each ToMM queue) if access to a single wait buffer is rate limiting. Second, the key of each entry in the Wait Buffer uniquely identifies the message for which it is waiting since the PNI is to prohibit a PE from having more than one outstanding reference to the same memory location.

After arriving at a FromMM port, a returning request, R-ret, is both routed to the appropriate ToPE queue and used to search associatively the relevant Wait Buffer. If a match occurs, the entry found, R-wait, is removed from the buffer and its function indicator, PE and MM numbers, and address are routed to the appropriate ToPE queue. If the request was a load, the data field is taken from R-ret; if a fetch-and-add, the R-wait data field is added to the R-ret data field.

To summarize the necessary hardware, we note that in addition to adders, registers, and routing logic, each switch requires two instances of each of the following memory units. For each unit we have indicated the operations it must support.

1. ToMM-queue: Entries are inserted and deleted in a queue-like fashion, associative searches may be performed, and matched entries may be updated.
2. ToPE-queue: Entries may be inserted and deleted in a queue-like fashion.
3. Wait-Buffer: Entries may be inserted and associative searches may be performed with matched entries removed.

Note that it is possible for more than two requests to be combined at a switch. However, the structure of the switch is simplified if it supports only combinations of pairs since a request returning from memory could then match at most one request in the Wait Buffer, eliminating the need for contention logic. Another advantage of not supporting multiple combinations within one switch is that it permits the pipelined implementation of the ToMM queue described below.

The switch can be partitioned into two essentially independent components, each implementing a unidirectional switch. The communication between the two components is restricted to the information pertaining to combined messages, that is, the information sent from the ToMM queues to the Wait Buffers. Since requests are combined relatively infrequently, the link between the two components can have a small bandwidth. We are currently investigating other possible partitions for a switch while noting that the its increased functionality impedes a bit-slice implementation.

3.3.1 _The ToMM Queue_ - As illustrated in Figure 4 our ToMM queue is an enhancement of the VLSI systolic queue of Guibas and Liang. We first describe the queue-like behavior of this structure and then explain how the necessary searching is accomplished.

Items added to the queue enter the middle column, check the adjacent slot in the right column, and move into this slot if it is empty. If the slot is full, the item moves up one position in the middle column and the process is repeated. (Should the item reach the top of the middle column and still be unable to shift right, the queue is full.) Meanwhile, items in the right column shift down, exiting the queue at the bottom. Before

* Although we use the term queue, entries within the middle of the queue may also be accessed.

** The design of the ToMM queue, permitting this search and subsequent actions to be performed with minimal delay, is detailed in setion 3.3.1.

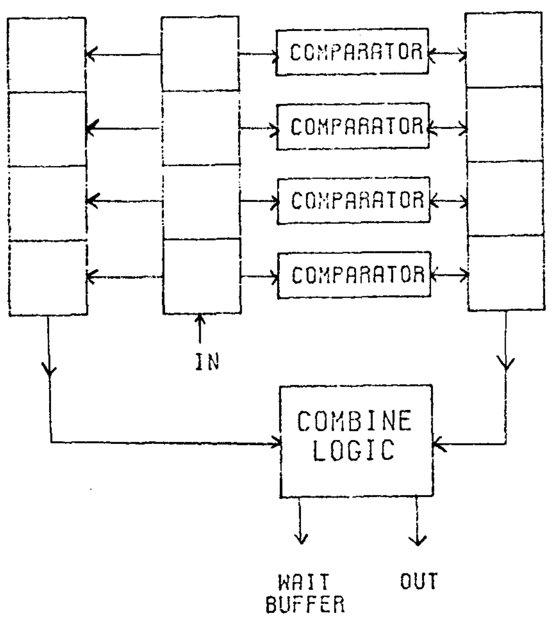

Figure 4. Systolic ToMM Queue

giving the enhancements needed for searching, we make four observations: the entries proceed in a FIFO order; as long as the queue is not empty and the switch in the next stage can receive an item, one item exits the queue at each cycle; as long as the queue is not full a new item can be entered at each cycle*; items are not delayed if the queue is empty and the next switch can receive them.

The queue is enhanced by adding comparison logic between adjacent slots in the right two columns, permitting a new entry moving up the middle column to be matched successively against all the previous entries as they move down the right column**. If a match is found, the matched entry moves (from the middle column) to the left column, called the "match column". Entries in the match column shift down at the same rate as entries on the right column of the queue. A pair of requests to be combined will therefore exit their respective columns at the same time and will thus enter the combining unit simultaneously.

Note that it is possible to reduce the width of the ToMM queue by having each request split into several successive entries. If requests are transmitted between switches as a series of successive packets, a ToMM queue with a width matching the size of these packets would avoid the assembly and disassembly of messages, resulting in a complete pipelining of the message processing. The smaller size of comparators and adders may also result in faster logic. A detailed description of the VLSI switch logic appears in Snir and Solworth [82].

*The number of cycles between successive insertions must, however, be even (zero included).

**Actually, an item is matched against half of the entries moving down the rigth column. This does not create any problem if a request consists of an even number of successive packets. If an entire request is contained in one packet then one needs either twice as many comparators or two cycles for each movement.

3.4 The Network Interfaces

The PNI (processor-network interface) performs four functions: virtual to physical address translation, assembly/disassembly of memory requests, enforcement of the network pipeline policy, and cache management. The MNI (memory-network interface) is much simpler, performing only request assembly/disassembly and the additions operation necessary to support fetch-and-add. Since the MNI operations as well as the first two PNI functions are straightforward, we discuss only pipelining policy and cache management.

Before detailing these two functions, we note two restrictions on pipelining memory requests (i.e. issuing a request before the previous one is acknowledged). As indicated above, pipelining requests indisciminately can violate the serialization principle (section 3.1.4), and furthermore, pipelining requests to the same memory location is not supported by our current switch design (3.3).

Since accessing central memory involves traversing a multistage network, effective cache management is very important. To reduce network traffic a write-back update policy was chosen: Writes to the cache are not written through to central memory; instead, when a cache miss occurs and eviction is necessary, updated words within the evicted block are written to central memory. Note that cache generated traffic can always be pipelined.

In addition to the usual operations described above, which are invisible to the PE, our cache provides two functions, release and flush, that must be specifically requested and can be performed on a segment level or for the entire cache. We now show that judicious use of release and flush further reduces network traffic.

The release command marks a cache entry as available without performing a central memory update. This enables a task to free cache space allocated to virtual addresses that will no longer be referenced. For example, private variables declared within a begin-end block can be released at block exit. Thus, the release operation reduces network traffic by lowering the quantity of data written back to central memory during a task switch. Moreover, if (prior to a task switch) another virtual address maps to a released cache address, no central memory update is necessary.

Release also facilitates caching shared read-write data during periods of read-only access: If a set of tasks sharing read-write data can guarantee that during a period of time no updates will occur, then the data is eligable for caching for the duration of this period. Subsequently, the data must be released and marked uncacheable to insure that no task uses stale data.

The flush facility, which enables the PE to force a write-back of cached values, is needed for task switching since a blocked task may be rescheduled on a different PE. To illustrate another use of flush and release, consider a variable V that is declared in task T and is shared with T's subtasks. Prior to spawning these subtasks, T may treat V as private (and thus eligible to be cached and pipelined) providing that V is flushed, released, and marked shared immediately before the subtasks are spawned. The flush updates main memory, the release insures that the parent task will not use stale data, and marking V shared enables T's subtasks to reference V. Once the subtasks have completed T may again consider V as private and eligable for caching. Coherence is maintained since V is cached only during periods of exclusive use by one task.

3.5 The Processors and Memory Modules

The MMs are standard components consisting of off the shelf memory chips. The PEs, however, need to be a (slightly) custom design since we require the fetch-and-add operation. Moreover, to fully utilize the high bandwidth connection network, a PE must continue execution of the instruction stream immediately after issuing a request to fetch a value from central memory. The target register would be marked "locked" until the requested value is returned from memory; an attempt to use a blocked register would suspend execution. Note that this policy is currently supported on large scale computers and is becoming available on one chip processors (Radin [82]). Software designed for such processors attempts to prefetch data sufficiently early to permit uninterrupted execution.

If the latency remains an impediment to performance, we would hardware-multiprogram the PEs (as in the CHOPP design (Sullivan [77]) and the Denelcor HEP machine (Denelcor [81]). Note that k-fold multiprogramming is equivalent to using k times as many PEs -- each having relative performance 1/k. Since, to attain a given efficiency, such a configuration requires larger problems, we view multiprogramming as a last resort.

Although we have not given sufficient attention to I/O, we have noticed that I/O processors can be substituted for arbitrary PEs in the system. More generally, since the design does not require homogeneous PEs, a variety of special purpose processors (e.g. FFT chips, matrix multipliers, voice generators, etc.) can be attached to the network.

3.6 Machine Packaging

We conservatively estimate that a machine built in 1990 would require four chips for each PE-PNI pair, nine chips for each MM-MNI pair (assuming a 1 megabyte MM built out of 1 megabit chips), and two chips for each 4-input-4-output switch (which replaces four of the 2x2 switches described above). Thus, a 4096 processor machine would require roughly 65,000 chips, not counting the I/O interfaces. Note that the chip count is still dominated, as in present day machines, by the memory chips, and that only 19% of the chips are used for the network. Nevertheless, most of the machine volume will be occupied by the network, and its assembly will be the dominant system cost, due to the nonlocal wiring required.

It is possible to partition an N input, N output Omega network built from 2x2 switches into sqrt(N) "input modules" and sqrt(N) "output modules". An input module consists of sqrt(N) network inputs and the sqrt(N)(log N)/4 switches that can be accessed from these inputs in the first (log N)/2 stages of the network. An output module consists of sqrt(N) network outputs and the sqrt(N)(log N)/4 switches that can be accessed from these outputs in the last half of the network. Moreover, it is possible to arrange the switches of each module so that, between any two successive stages, all lines have the same length (Figure 5).

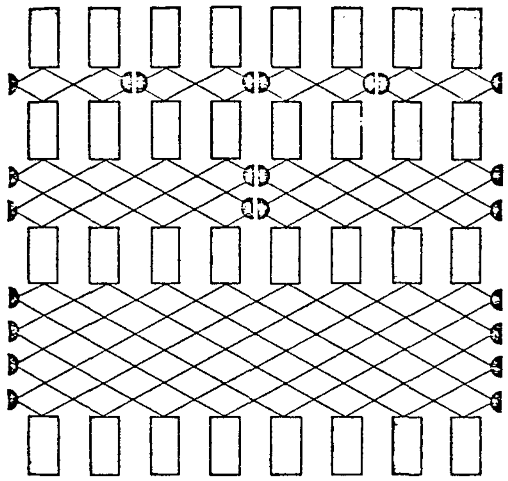

Figure 5. Layout of Network on Boards

Finally, if the input boards are stacked vertically on one rack, the output boards are stacked vertically on another rack, and the two racks are stacked one atop another, such that the boards on one rack are orthogonal to the boards on the other rack, then all off board lines will run nearly vertically between the two sets of boards as illustrated in Figure 6 (Figures 5 and 6 are reprinted from Wise [81]).
The same strategy can be used for networks built of kxk switches.

We propose using this layout for a 4K processor machine constructed from the chips described at the beginning of this section. This

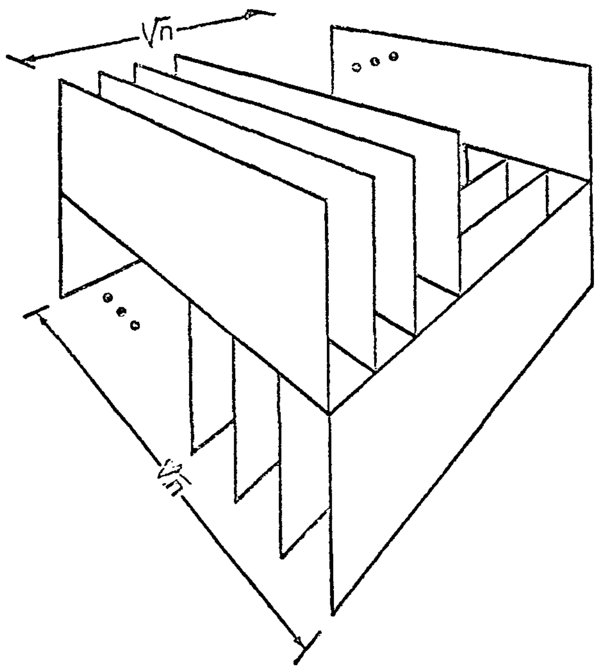

Figure 6. Packaging of Network Boards

machine would include two types of boards: "PE boards" that contain the PEs, the PNIs, and the first half of the network stages and "MM boards" that contain the MMs, the MNIs and the last half of the network stages. Using the chip counts given above, a 4K PE machine built from two chip 4x4 switches would need 64 PE boards and 64 MM boards, with each PE board containing 352 chips and each MM board containing 672 chips. Since the PE chips will be near the free edge of the PE board and the MM chips will be near the free edge of the MM board, I/O interfaces can be connected along these edges.

4.0 COMMUNICATION NETWORK PERFORMANCE

Since the overall ultracomputer performance is critically dependent on the communication network and this network is likely to be the most expensive component of the completed machine, it is essential to evaluate the network performance carefully so as to choose a favorable configuration.

4.1 Performance Analysis

Although each switch in the network requires a significant amount of hardware it appears feasible to implement a 2x2 switch on one chip, using today's technology. Further, we assume it will be feasible in 1990 technology to implement 4x4, or even 8x8 switches on one chip. It seems, however, that the main restriction on the switch performance will be the rate at which information can be fed into and carried from the chip, rather than the rate at which that information can be processed within the chip. The basic hardware constraint will be, therefore, the number of bits that can be carried on or off the chip in one unit of time (one cycle).

Suppose that 400 bits can be transferred on or off the chip in one cycle (which we estimate, for 1990 technology, to be on the order of 25 nsec). If each message transmitted through the network consists of approximately 100 bits (64 bits data, 30 bits address), then a 2x2 switch needs two cycles for the transfer of the 800 bits involved in the relaying of two messages in each direction. It is, however, possible to pipeline the transmission of each message, so that the delay at each switch is only one cycle if the queues are empty.

The chip bandwidth constraint does not determine a unique design for the network. It is possible to replace 2x2 switches by kxk switches, time multiplexing each line by a factor of k/2. It is also possible to use several copies of the same network, thereby reducing the effective load on each one of them and enhancing network reliability. We present performance analyses of various networks in order to indicate the tradeoffs involved.

A particular configuration is characterized by the values of the following three parameters:

1. k - the size of the switch. Recall that a kxk switch requires 4k lines.
2. m - the time multiplexing factor, i.e. the number of switch cycles required to input a message (to simplify the analysis we assume that all the messages have the same length).
3. d - the number of copies of the network that are used.

The chip bandwidth constraint yields an upper bound on the k/m ratio. We shall assume therefore that this ratio is a constant B for all designs. Note that for any k a network with n inputs and n outputs can be built from (n lgn)/(k lgk) kxk switches and a proportional number of wires. Since our network contains a large number of identical switches, the network's cost is essentially proportional to the number of switches and independent of their complexity. We thus define the cost of a configuration to be C*(n lgn), where the cost factor C = d/(k lgk) (we are neglecting the small cost of interfacing several copies of the network).

In order to obtain a tractible mathematical model of the network we have made the following simplifying assumptions:

1. Requests are not combined.
2. Requests have the same length.

3. Queues are of infinite size.

4. Requests are generated at each PE by independent identically distributed time-invariant random processes.

5. MMs are equally likely to be referenced.

Let p be the average number of messages entered into the network by each PE per network cycle. If the queues at each switch are large enough ("infinite queues") then the average switch delay is approximately 1 + m↑2*p(1-1/k)/2(1-mp) cycles (see Kruskal and Snir; similar results can be found in Jacobsen and Misunas [77], and in Dias and Jump [81]). The average network traversal time (in one direction) is the number of stages times the switch delay plus the setting time for the pipe. Thus the number of cycles is:

T = (lgn/lgk)(1 + m↑2*p(1-1/k)/2(1-mp) + m - 1.

Let us note, however, the following facts:

1. The average number of messages per cycle entered into the network by each PE, p, must be smaller than 1/m, as it takes m cycles to input a message. Conversely, the network has a capacity of 1/m messages per cycle per PE, that is it can accommodate any traffic below this threshold. Thus, the global bandwidth of the network is indeed proportional to the number of PEs connected to it.

2. The initial 1 in the expression for the switch delay corresponds to the time required for a message to be transmitted through a switch without being queued (the switch service time). The second term corresponds to the average queueing delay. This term decreases to zero when the traffic intensity p decreases to zero and increases to infinity when traffic intensity p increases to the 1/m threshold. The surprising feature of this formula is the m↑2 factor, which is explained by noting that the queueing delay for a switch with a multiplexing factor of m is roughly the same as the queueing delay for a switch with a multiplexing factor of one, a cycle m times longer, and m times as much traffic per cycle.

We now use these formulae to compare the performance of different configurations. Let us assume that, using kxk switches, a time multiplexing factor m = k, that is, the bandwidth constant B = 1. Using d copies of the network reduces the effective load on each copy by a factor of d. Thus the average transit time for a network consisting of d Omega-networks composed of kxk switches is

T = (1 + k(k-1)p/2(d-kp))lgn/lgk + k-1 cycles,

where p is, as before, the average number of messages sent to the network by each PE per cycle. As expected, delays decrease when d increases. The dependency on k is more subtle. Increasing k decreases the number of stages in the network, but increases the pipelining factor, and therefore increases the queuing delays and the pipe setting delay.

We have plotted in Figure 7 the graphs of T as a function of the traffic intensity, p, for different values of k and d. We see that for reasonable traffic intensities (see next paragraph) a duplexed network composed of 4x4 switches yields the best performance.

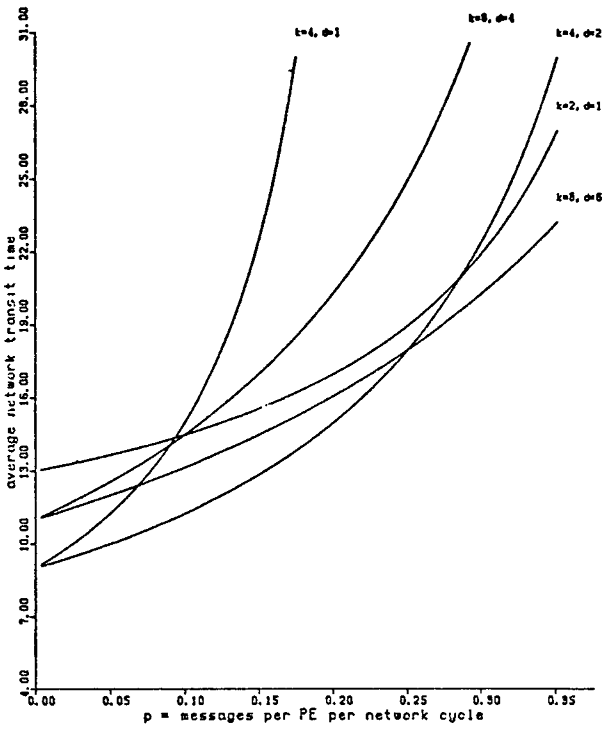

Figure 7.
Transit Times for Different Configurations

A network with 8x8 switches and d=6 also yields an acceptable performance, at approximately the same cost as the previous network. Since the bandwidth of the first network is d/k=.5 and the bandwidth of the second is .75, we see that for a given traffic level the second network is less heavily loaded and thus should provide better performance for traffic with high variance.

The above discussion indicates the type of considerations involved in the choice of an optimal configuration. An actual choice requires more accurate assessments of the technological constraints and the traffic distribution. The pipelining delays incurred for large multiplexing factors, the complexity of large switches, and the heretofore ignored cost and performance penalty incurred with interfacing many network copies, will probably make the use of switches larger than 8x8 impractical for a 4K PE parallel machine.

The previous discussion assumed a one chip implementation of each switch. By using the two chip implementation described at the end of section

3.3, one can nearly double the bandwidth of each switch while doubling the chip count. As delays are highly sensitive to the multiplexing factor m, this implementation would yield a better performance than that obtained by taking two copies of a network built of one chip switches. (It would also have the extra advantage of decreasing the gate count on each chip.) Thus, the ultimate choice may well be one network built of 4x4 switches, each switch consisting of two chips.

We now return to the five assumptions listed above, all of which may not be satisfied by actual hardware implementations. Our first two assumptions, that all messages are of equal (maximal) length and traverse the entire network, are clearly conservative: In practice, messages that do not carry data (load requests and store acknowledgements) would be shorter and merged messages do not each traverse the entire network.

Simulations have shown that queues of modest size (≤ 8) gives essentially the same performance as infinite queues. Although the requests generated by PEs cooperating on a single problem are not independent, the presence of a large number of PEs and a number of different problems will tend to smooth the data. On the other hand, even in a large system the pattern of requests by a single PE will be time dependent and further analytic and simulation studies are needed to determine the effect of this deviation from our assumed model.

Finally, by applying a hashing function when translating from virtual to physical addresses, the system can ensure that each MM is equally likely to be referenced.

4.2 Network Simulations

Our discussion of the possible configurations for the communication network still lacks two essential ingredients: an assessment of the traffic intensity we expect to encounter in practical applications, and an assessment of the impact of the network delay on the overall performance.

We routinely run parallel scientific programs under a paracomputer simulator (see Gottlieb [80c]) to measure the speedup obtained by parallelism and to judge the difficulty involved in creating parallel programs (see section 5). A recent modification allows us to simulate an approximation to the proposed network design rather than an ideal paracomputer: Since an accurate simulation would be very expensive, we used instead a multi-stage queuing system model with stochastic service time at each stage (see Snir [81]), parameterized to correspond to a network with six stages of 4x4 switches, connecting 4096 PEs to 4096 MMs. A message was modeled as one packet if it did not contain data and as three packets otherwise. Each queue was limited to fifteen packets and both the PE instruction time and the MM access time were assumed to equal twice the network cycle time. Thus the minimum central memory access time, which consists of the MM access time plus twice the the minimum network transit time, equals eight times the PE instruction time.

We have monitored the amount of network traffic generated by several scientific programs under the pessimistic assumptions that no shared data is cached and the optimistic assumption that all references to program text and private data are satisfied by the cache. The programs studied were:

1. A parallel version of part of a NASA weather program (solving a two dimensional PDE), with 16 PEs.

2. The same program, with 48 PEs.

3. The TRED2 program described in section 5, with 16 PEs.

4. A multigrid Poisson PDE solver, with 16 PEs.

Table 1 summarizes simulations of the four previously mentioned programs. The time unit is the PE instruction time.

	avg. CM access time	idle cycles	idle cycles per CM load	memory ref per instr	shared ref per instr
1	8.94	37%	5.3	0.21	.08
2	8.83	39%	4.5	0.19	.08
3	8.81	22%	4.9	0.25	.05
4	8.85	19%	3.5	0.24	.06

Table 1. Network Traffic and Performance

In these simulations the number of requests to central memory (CM) are comfortably below the maximal number that the network can support and indeed the average access time is close to the minimum. (Since each PE was a CDC 6600-type CPU, most instructions involved register-to-register transfers.) Specifically, only one instruction every five cycles for the first two programs (and one every four for the last two) generated a data memory reference.* Moreover only one data memory reference out of 2.6 in the first two programs, and one reference out of five for the last two programs were for shared data. We note that the last two programs were designed to minimize the number of accesses to shared data. As a result the number of idle cycles was significantly higher for the first two programs. Since the code generated by the CDC compiler often prefetched operands from memory, the average number of idle cycles per load from average central memory was significantly lower than the central memory access time.

* Since for the first two programs, the PEs were idle (waiting for a memory reference to be satisfied) approximately 40% of the time, five cycles corresponds to approximately three instructions.

We conclude that were these studies repeated on actual hardware the traffic intensity would be low ($p<.04$), and prefetching would mitigate the problem of large memory latency. The first conclusion, however, must be strongly qualified. The simulator we used is much less sensitive to fluctuations in the network traffic than an actual network would be. Moreover, we have ignored both cache generated traffic and the effect of operating system programs.

5.0 SIMULATION AND SCIENTIFIC PROGRAMMING

As indicated above we use an instruction level paracomputer simulator to study parallel variants of scientific programs. Applications already studied include radiation transport, incompressible fluid flow within an elastic boundary, atmospheric modeling, and Monte Carlo simulation of fluid structure. Current efforts include both extending the simulator to model the connection network more faithfully and running programs under a parallel operating system scheduler.

The goals of our paracomputer simulation studies are, first, to develop methodologies for writing and debugging parallel programs and second, to predict the efficiency that future large scale parallel systems can attain. As an example of the approach taken, and of the results thus far obtained, we report on experiments with a parallelized variant of the program TRED2 (taken from Argonne's EISPACK library), which uses Householder's method to reduce a real symmetric matrix to tridiagonal form (see Korn [81] for details).

An analysis of the parallel variant of this program shows that the time required to reduce an N by N matrix using P processors is well approximated by

$$T(P,N) = aN + dN{\uparrow}3/P + W(P,N)$$

where the first term represents "overhead" instructions that must be executed by all PEs (e.g. loop initializations), the second term represents work that is divided among the PEs, and $W(P,N)$, the waiting time, is of order $\max(N,P{\uparrow}.5)$. We determined the constants experimentally by simulating TRED2 for several (P,N) pairs and measuring both the total time T and the waiting time W. (Subsequent runs with other (P,N) pairs have always yielded results within 1% of the predicted value.) Table 2 summarizes our experimental results and supplies predictions for problems and machines too large to simulate (these values appear with an asterisk). In examining this table, recall that the efficiency of a parallel computation is defined as

$$E(P,N) = T(1,N)/(P*T(P,N)) .$$

\ PE N \	Reduction of Matrices to Tridiagonal Form				
	16	64	256	1024	4096
16	62%	26%	7%	1%*	0%*
32	87%	60%	25%	6%*	1%*
64	96%	86%	59%	27%*	7%*
128	99%*	96%*	86%*	59%*	24%*
256	100%*	99%*	96%*	86%*	58%*
512	100%*	100%*	99%*	96%*	85%*
1024	100%*	100%*	100%*	99%*	96%*

Table 2. Measured and Projected Efficiencies.

Although we consider these measured efficiencies encouraging, we note that system performance can probably be improved even more by sharing PEs among multiple tasks. (Currently the simulated PEs perform no useful work while waiting.) If we make the optimistic assumption that all the waiting time can be recovered, the efficiencies rise to the values given in Table 3.

\ PE N \	Reduction of Matrices to Tridiagonal Form				
	16	64	256	1024	4096
	(without waiting time)				
16	71%	37%	12%	3%	0%
32	90%	69%	35%	12%	3%
64	97%	90%	68%	35%	12%
128	99%	97%	90%	68%	35%
256	100%	99%	97%	90%	68%
512	100%	100%	99%	97%	90%
1024	100%	100%	100%	99%	97%

Table 3. Projected Efficiencies.

6.0 CONCLUSION

Our simulations have conclusively shown that a paracomputer containing thousands of processors would be an extremely powerful computing engine for large scientific programs. But such ideal machines cannot be built. In this report we have described a realizable approximation, the NYU Ultracomputer. We believe that, within the decade, a 4096 PE Ultracomputer can be constructed with roughly the same component count as found in today's large machines. Although our Ultracomputer simulations are still fragmentary, the preliminary results thus far obtained are encouraging.

To demonstrate further the feasibility of the hardware and software design, we plan to construct an 8 PE and subsequently a 64 PE prototype using the switches and interfaces described above to connect commercial microprocessors and memories.

APPENDIX

Management of Highly Parallel Queues

Since queues are a central data structure for many algorithms, a concurrent queue access method can be an important tool for constructing parallel programs. In analyzing one of their parallel shortest path algorithms, Deo et al. [80] dramatize the need for this tool:

"However, regardless of the number of processors used, we expect that algorithm PPDM has a constant upper bound on its speedup, because every processor demands private use of the Q."

Refuting this pessimistic conclusion, we show in this appendix that, although at first glance the important problem of queue management may appear to require use of at least a few inherently serial operations, a queue can be shared among processors without using any code that could create serial bottlenecks. The procedures to be shown maintain the basic first-in first-out property of a queue, whose proper formulation in the assumed environment of large numbers of simultaneous insertions and deletions is as follows: If insertion of a data item p is completed before insertion of another data item q is started, then it must not be possible for a deletion yielding q to complete before a deletion yielding p has started.

In the algorithm below we represent a queue of length Size by a public circular array Q[0:Size-1] with public variables I and D pointing to the locations of the items last inserted and deleted (these correspond to the rear and front of the queue respectively). Thus MOD(I+1,Size) and MOD(D+1,Size) yield the locations for the next insertion and deletion, respectively. Initially I=D=0 (corresponding to an empty queue).

We maintain two additional counters, #Ql and #Qu, which hold lower and upper bounds respectively for the number of items in the queue, and which never differ by more than the number of active insertions and deletions. Initially #Ql=#Qu=0, indicating no activity and an empty queue. The parameters QueueOverflow and QueueUnderflow appearing in the program shown below are flags denoting the exceptional conditions that occur when a processor attempts to insert into a full queue or delete from an empty queue. (Since a queue is considered full when #Qu $\geq$ Size and since deletions do not decrement #Qu until after they have removed their data, a full queue may actually have cells that could be used by another insertion.) The actions appropriate for the QueueOverflow and QueueUnderflow conditions are application dependent: One possibility is simply to retry an offending insert or delete; another possibility is to proceed to some other task.

Critical-section-free Insert and Delete programs are given below. The insert operation proceeds as follows: First a test-increment-retest (TIR) sequence is used to guarantee the existence of space for the insertion, and to increment the upper bound #Qu. If the TIR fails, a QueueOverflow occurs. If it succeeds, the expression Mod(Fetch<u>A</u>dd(I,1),Size) gives the appropriate location for the insertion, and the insert procedure waits its turn to overwrite this cell (see Gottlieb et al. [81]). Finally, the lower bound #Ql is incremented. The delete operation is performed in a symmetrical fashion; the deletion of data can be viewed as the insertion of vacant space.

```
Procedure Insert(Data,Q,QueueOverflow)
        If TIR(#Qu,1,Size) Then  {
                MyI <-- Mod(FetchAdd(I,1),Size)
                Wait turn at MyI
                Q[MyI] <-- Data
                FetchAdd(#Ql,1)
                QueueOverflow <-- False  }
        Else    QueueOverflow <-- True
End Procedure

Procedure Delete(Data,Q,QueueUnderflow)
        If TDR(#Ql,1) Then  {
                MyD <-- Mod(FetchAdd(D,1),Size)
                Wait turn at MyD
                Data <-- Q[MyD]
                FetchAdd(#Qu,-1)
                QueueUnderflow <-- False  }
        Else    QueueUnderflow <-- True
End Procedure

Boolean Procedure TIR(S,Delta,Bound)
        If S+Delta ≤ Bound  Then
                If FetchAdd(S,Delta) ≤ Bound Then
                         TIR <-- true
                Else  {  FetchAdd(S,-Delta)
                         TIR <-- false }
End Procedure

Boolean Procedure TDR(S,Delta)
        If S-Delta ≥ 0  Then
                If FetchAdd(S,-Delta) ≥ 0 Then
                         TDR <-- True
                Else  {  FetchAdd(S,Delta)
                         TDR <-- false  }
End Procedure
```

Although the initial test in both TIR and TDR may appear to be redundant, a closer inspection shows that their removal permits unacceptable race conditions. This point is also expanded in Gottlieb et al. [81] where other fetch-and-add based software primatives are presented as well.

It is important to note that when a queue is neither full nor empty our program allows many insertions and many deletions to proceed completely in parallel with no serial code executed. This should be contrasted with current parallel queue algorithms, which use small critical sections to update the insert and delete pointers.

REFERENCES

A. Borodin and J. E. Hopcroft, "Merging on Parallel Models of Computation", Manuscript, 1981.

Burroughs Corp., Numerical Aerodynamic Simulation Facility Feasibilty Study, NAS2-9897, March 1979.

Denelcor, Heterogeneous Element Processor Principles of Operation, 1981.

Narsingh Deo, C. Y. Pang, and R. E. Lord, "Two Parallel Algorithms for Shortest Path Problems"

Daniel Dias and J. Robert Jump, "Analysis and simulation of buffered delta networks", IEEE Trans. on Computers C-30, 1981, 273-282.

L. Rodney Goke and G.J. Lipovsky, "Banyan Networks for Partitioning Multiprocessor Systems", First Annual Symp. on Computer Architecture, 1973, 21-28.

Allan Gottlieb, "PLUS - A PL/I Based Ultracomputer Simulator, I", Ultracomputer Note #10, Courant Institute, NYU 1980a.

Allan Gottlieb, "WASHCLOTH - The Logical Successor to Soapsuds", Ultracomputer Note #12, Courant Institute, NYU 1980b.

Allan Gottlieb, "PLUS - A PL/I Based Ultracomputer Simulator, II", Ultracomputer Note #14, Courant Institute, NYU 1980c.

Allan Gottlieb and Clyde P. Kruskal, "MULT - A Multitasking Ultracomputer Language with Timing, I & II", Ultracomputer Note #15, Courant Institute, NYU 1980.

Allan Gottlieb and Clyde P. Kruskal, "Coordinating Parallel Processors: A Partial Unification", Computer Architecture News, October 1981, pp. 16-24.

Allan Gottlieb, Boris Lubachevsky, and Larry Rudolph, "Coordinating Large Numbers of Processors", Intl. Conf. on Parallel Procesing, 1981.

Allan Gottlieb and Jack T. Schwartz, "Networks and Algorithms for Very Large Scale Parallel Computations", to appear in Computer, January 1982.

Leo J. Guibas and Frank M. Liang, "Systolic stacks, queues, and counters", Conference on Advanced Research in VLSI, Jan. 1982.

Robert G. Jacobsen and David P. Misunas, "Analysis of structures for packet communication", Int. Conf. on Parallel Processing, 1977.

Steven D. Johnson, "Connection Networks for Output-Driven List Multiprocessing", Tech. Rep. 114, Computer Science Dept., Indiana University, 1981.

Malvin Kalos, "Scientific Calculations on the Ultracomputer", Ultracomputer Note #30, Courant Institute, NYU, 1981.

Malvin Kalos, Gabi Leshem, and B. D. Lubachevsky, "Molecular Simulations of Equilibrium Properties", Ultracomputer Note #27, Courant Institute, NYU, 1981.

K. R. Kaplan and R. V. Winder, "Cache-Based Computer Systems", Computer 6, 1973, pp. 30-36.

David Korn, "Timing Analysis for Scientific Codes Run under WASHCLOTH Simulation", Ultracomputer Note #24, Courant Institute, NYU, 1981.

Clyde P. Kruskal, "Supersaturated Paracomputer Algorithms", Ultracomputer Note #26, Courant Institute, NYU, 1981.

Clyde P. Kruskal and Marc Snir, "Some results on Packet-Switching Networks for Multiprocessing", Princeton Conference on Information Sciences and Systems, 1982.

H. T. Kung, "The Structure of Parallel Algorithms", in Advances in Computers 19, M. C. Yovits (ed.), Academic Press, New York, 1980, 65-112.

Leslie Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", IEEE Trans. C-28, 1979, pp. 690-691.

Duncan Lawrie, "Access and Alignment of Data in an Array Processor", IEEE Trans. C-24, 1975, pp. 1145-1155.

Gary Rodrigue, E. Dick Giroux, and Michael Pratt, "Perspectives on Large-Scale Scientific Computing", IEEE Computer v. 13 #10, Oct. 1980, pp. 65-80.

Norman Rushfield, "Atmospheric Computations on Highly Parallel MIMD Computers", Ultracomputer Note #22, Courant Institute, NYU, 1981

J. T. Schwartz, "Preliminary Thoughts on Ultracomputer Programming Style", Ultracomputer Note #3, Courant Institute, NYU, 1979.

J. T. Schwartz, "Ultracomputers", ACM TOPLAS, 1980, pp. 484-521.

Howard J. Siegel and Robert J. McMillen, "Using the Augmented Data Manipulator Network in PASM", Computer, 14, 1981, pp. 25-34.

Burton J. Smith, "A Pipelined, Shared Resource MIMD Computer", Intl. Conf. on Parallel Processing, 1978, pp. 6-8.

Marc Snir, "'NETSIM' Network Simulator for the Ultracomputer", Ultracomputer Note #28, Courant Institute, NYU, 1981.

Marc Snir and Jon Solworth, "The Ultraswitch -- A VLSI Network Node for Parallel Processing", Ultracomputer Note #39, Courant Institute, NYU, 1982.

Harold S. Stone, "Parallel Computers", in _Introduction to Computer Architecture_, Harold S. Stone (ed.), SRA, Chicago Ill., 1980, pp. 318-374.

Herbert Sullivan, Theodore Bashkow, and David Klappholz, "A Large Scale Homogeneous, Fully Distributed Parallel Machine", _Proc. of the 4th Annual Sump. on Comp. Arch._, 1977, pp. 105-124.

R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm* - A Modular, Multi-microprocessor", _Proc. AFIPS Conf._ 46, 1977, pp. 637-644.

D. S. Wise, "Compact Layout of Banyan/FFT Networks", _CMU Conf. on VLSI systems and Computations_, Kung, Sproull and Steele (eds.), Computer Science Press, Rockville Maryland, 1981, pp. 186-195.