An Overview of the NYU Ultracomputer Project

by

Allan Gottlieb

Ultracomputer Note #100
July, 1986
Revised: October 1986
Revised: April 1987
Revised: October 1987

ABSTRACT

The NYU Ultracomputer is a shared memory MIMD parallel computer design to contain thousands of processors connected by an Omega network to a like number of memory modules. A new coordination primitive fetch-and-add is defined and the network is enhanced to combine simultaneous requests, including fetch-and-adds, directed at the same memory location. The present report, an edited amalgam of papers I have coauthored with other members of the Ultracomputer project, discusses our work on architecture, programming models, system software, hardware and VLSI design, performance analysis, and simulation studies.

This work was supported in part by the Applied Mathematical Sciences Program of the US Department of Energy under contract DE-AC02-76ER03077 and grant DE-FG02-88ER25052 and in part by the National Science Foundation under grants DCR-8413359 and DCR-8320085.

1. Introduction

Since its inception in 1979, the NYU Ultracomputer project has studied the problem of how the fruits of the ongoing microelectronic revolution could be best used to solve problems having computational demands so vast as to make them inaccessible using current technology. Like others, we have always believed that parallelism provides the key new ingredient. With the promise that in 1990 10-20 MIPS (including floating point instructions) and several megabytes of memory would be available on a handful of chips producing just a few watts, one is naturally led to consider ensembles of thousands of processor with gigabytes of memory[1]. Such a configuration would yield several orders of magnitude more performance than do current supercomputers from roughly the same component count. Moreover, the number of distinct components would be quite small and thus the design appears attractive.

However, it remains to be demonstrated that this potentially high-performance assemblage can be effectively utilized. There are two dimensions to the challenge. First, several thousand processors must be coordinated in such a way that their aggregate power is applied to useful computation. Serial procedures in which one processor works while the others wait become bottlenecks that drastically reduce the power obtained. Indeed, for any highly parallel architecture, the relative cost of a serial bottleneck rises linearly with the number of processors involved. Second, the machine must be programmable by humans. Effective use of high degrees of parallelism will be facilitated by simple languages and facilities for designing, writing, and debugging parallel programs.

We propose that the hardware and software design of a highly parallel computer should meet the following goals.

- Scaling. Effective performance should scale upward to a very high level. Given a problem of sufficient size, an n-fold increase in the number of processors should yield a speedup factor of almost n.
- General purpose. The machine should be capable of efficient execution of a wide class of algorithms, displaying relative neutrality with respect to algorithmic structure or data flow pattern.
- Programmability. High-level programmers should not have to consider the machine's low-level structural details in order to write efficient programs. Programming and debugging should not be substantially more difficult than on a serial machine.
- Multiprogramming. The software should be able to allocate processors and other machine resources to different phases of one job and/or to different user jobs in an efficient and highly dynamic way.

Achieving these goals requires an integrated hardware/software approach. The burden on the system designer is to support a programming model that is high-level and flexible, to schedule the processors so that the workload is balanced, and, most importantly, to avoid inherently serial program sections that

[1]Curiously, the fastest computer of the day always seems to contain about 10^5 pieces and dissipates about 10^5 watts. This is one reason why we do not propose a million processor ultracomputer; designs featuring millions of processor always specify smaller, less complex computing engines, so that many can be packed onto a single chip.

would constitute unacceptable serial bottlenecks.

We will consider these and subsidiary issues primarily in the context of a particular parallel architecture, the NYU Ultracomputer, but, when considering software issues, will also discuss the newer IBM RP3 design, which includes the Ultracomputer architecture as well as significant IBM enhancements, and for which we are designing and implementing the operating system. The Ultracomputer machine design, which will be detailed below, was driven by the above objectives plus the desire that the machine be buildable without requiring unanticipated advances in hardware technology or packaging.

It is interesting to recall that the original Ultracomputer, described in Schwartz [80], was a rather different architecture, without shared memory and using largely synchronous communication. Contacts with the Burroughs team proposing a design for NASA's Numerical Aerodynamic Simulation Facility led us to consider MIMD shared memory architectures more seriously. We had been using high level simulators (Gottlieb [80a, 80c] and Gottlieb and Kruskal [80]) of the Schwartz Ultracomputer and found the shared memory MIMD design easier to program. During the next several months we found hardware and software enhancements to the latter architecture and adopted it.

After describing the machine model in the next section, we turn to software issues, emphasizing the different programming styles possible on the Ultracomputer, the task scheduling and memory management disciplines they suggest, and the system software support needed. A long section describes the machine itself, concentrating on the network and the ongoing VLSI design of the enhanced switch that permit the network to combine memory references. The next two sections discuss analytic and simulation studies of the network and our early work on writing and executing scientific programs on an instruction level simulator. We conclude by giving a status report of our prototype hardware, system software, and custom VLSI chips, and our future plans.

The material to be presented represents research conducted by the entire NYU Ultracomputer project and not just the present author. Indeed the report itself is an edited amalgam of papers that I have coauthored with Susan Dickey, Jan Edler, Ralph Grishman, Richard Kenner, Clyde P. Kruskal, Jim Lipkis, Kevin P. McAuliffe, Larry Rudolph, Marc Snir, Patricia J. Teller, and Jim Wilson. The specific references are Dickey *et al.* [86a], Edler *et al.* [85a, 85b, 86], and Gottlieb *et al.* [83a].

2. Machine Model

In this section we first review the paracomputer or WRAM 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 shared memory computational model we examine several alternative models and justify our selection. Although the model to be described is not physically realizable, we shall see in sub-

sequent sections that an MIMD[2] shared memory machine closely approximating our idealized model can indeed be built.

Our experience indicates that the model chosen can lead to an easily-programmable highly-parallel computer. This experience includes implementing several significant (scientific) applications under such a model, both by modifying existing serial programs and by writing programs ‘‘from scratch’’ (Kalos [81]). In all cases the incremental effort required to obtain a parallel program was not excessive, always significantly less than what was needed to obtain an equivalent serial program. When the referenced applications were run under simulation, the results indicated that for large problems, speedups nearly linear in the number of processors (PEs) are obtained, thus indicating that the performance of this model scales upward to very high levels.

2.1. The Idealized Paracomputer or WRAM

An idealized parallel processor, dubbed a ‘‘paracomputer’’ by Schwartz [80] and classified as a WRAM by Borodin and Hopcroft [82], consists of a number of autonomous processing elements (PEs) sharing a central memory[3]. 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.

In order to make precise the effect of simultaneous access to shared memory we define the *serialization principle* (cf. Eswaran et al. [76]), which states that the effect of simultaneous actions by the PEs is as if the actions had occurred in some (unspecified) serial order. Thus, for example, a load simultaneous with two stores directed at the same memory cell will return either the original value or one of the two stored values, possibly different from the value that the cell finally 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.

Several points need to be noted concerning the difficulty of a hardware implementation of this model. First, the single cycle access to globally shared memory is not possible to achieve. Indeed, for any technology there is a limit, say *b*, on the number of signals that one can fan in at once. Thus, if N processors are to access even a single bit of shared memory, the shortest access time possible is $\log_b N$. As will be seen, hardware closely approximating this behavior has been designed. But it does not come for free. The processor to memory interconnection network used cannot be effectively constructed using off the shelf components; a custom VLSI design is needed. In addition to increasing the design time for

[2]In the taxonomy of Flynn [66], MIMD is the category of Multiple Instruction stream, Multiple Data stream computers; whereas in SIMD designs there is just a Single Instruction stream for the Multiple Data streams.

[3]To be more specific, Fortune and Wyllie [78] introduced the PRAM (Parallel Random Access Machine), a multiple processor analogue of Aho *et al.*’s [74] RAM, in which concurrent reads of a single location are permitted but concurrent writes are not. Snir [82] classifies this machine as a CREW PRAM (Concurrent Read Exclusive Write). Both the WRAM and the paracomputer also permit concurrent writes and are thus classified as CRCW PRAMs by Snir. Our model is essentially a CRCW PRAM augmented by the fetch-and-add coordination primitive defined below.

such a machine, the network adds to its replication cost and size. Thus, for a fixed number of dollars (or cubic feet, or BTUs, etc.), a shared memory design will contain fewer processors or memory cells than will a strictly private memory design constructed from the same technology. Although we believe that the lower peak performance inherent in shared memory designs is adequately compensated for by their increased flexibility and generality, this issue is not settled. Most likely the answer will prove to be so application dependent that both shared and private memory designs will prove successful. A similar tradeoff between peak performance and flexibility arises in the choice of autonomous processors, i.e. an MIMD design. The alternative SIMD architecture class, in which for any given cycle, all active processors execute the same instruction, has already proven itself to be a cost-effective method of attaining very high performance on suitable applications. Indeed, all current vector supercomputers fall into this class. It is also well known, however, that converting algorithms to vector form is often difficult and the resulting programs tend to be brittle, i.e. hard to transport to different hardware or to modify to use improved algorithms.

2.2. The Fetch-and-add Operation

We augment our computational model with the ''fetch-and-add'' operation, a simple yet highly effective interprocessor coordination[4] operation that permits highly concurrent execution of operating system primitives and application programs (see Gottlieb and Kruskal [81]). Fetch-and-add is essentially an indivisible add to memory; its format is F&A(V,e), where V is an integer variable and e is an integer expression. The operation is defined to return the (old) value of V and to replace V by the sum V+e. Moreover, concurrent fetch-and-adds are required to satisfy the serialization principle enunciated above. Thus, fetch-and-add operations simultaneously directed at V would cause V to be modified by the appropriate total increment while each operation yields the intermediate value of V corresponding to its position in an equivalent serial order. The following example illustrates the semantics of fetch-and-add: Assume V is the only shared variable. If PE_i executes

$$Y_i \longleftarrow F\&A(V,e_i) \ ,$$

and if PE_j simultaneously executes

$$Y_j \longleftarrow F\&A(V,e_j) \ ,$$

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

$$Y_i \longleftarrow V$$
$$Y_j \longleftarrow V+e_i$$

or

[4]The term synchronization is more commonly used but we believe coordination is more accurate.

$$Y_i \longleftarrow V+e_j$$
$$Y_j \longleftarrow V$$

and, in either case, the value of V becomes $V+e_i+e_j$.

For another example consider several PEs concurrently executing F&A(I,1), where I is a shared variable used to index into a shared array. Each PE obtains an index to a distinct array element (although one cannot predict which element will be assigned to which PE), and I receives the appropriate total increment.

In a later section we present 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. This design can be extended to support the more general fetch-and-ϕ and certain RMW operations described below.

2.3. The General Fetch-and-ϕ Operation

Fetch-and-add is a special case of the more general fetch-and-ϕ operation, where ϕ may be any associative binary operator, introduced by Gottlieb and Kruskal [81]. This operation fetches the value in V and replaces it with ϕ(V,e). If ϕ is, in addition, commutative, the final value of V, after the completion of concurrent fetch-and-ϕ's, is independent of the serialization order chosen. Of course defining $\phi(a,b)=a+b$ gives fetch-and-add.

We now show that two important coordination primitives, swap and test-and-set, may also be obtained as special cases of fetch-and-ϕ. (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:

```
T&S(V)
   { Temp ⟵ V
   V ⟵ TRUE }
   RETURN Temp
```

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 the Boolean Or function gives test-and-set, specifically

T&S(V) is equivalent to F&Or(V,TRUE).

Similarly, a swap operation can be effected by using the projection operator π_2, where $\pi_2(a,b) = b$; i.e.

Swap(L,V) is equivalent to $L \longleftarrow F\&\pi_2(V,L)$.

We conclude the discussion of fetch-and-ϕ 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-ϕ. To load the local variable L from a variable V stored in central memory one simply executes

$$L \longleftarrow F\&\pi_1(V,*)$$

where $\pi_1(a,b)=a$ and the value of $*$ is immaterial (and thus need not be transmitted). Similarly, to store the value of L into V one executes

$$* \longleftarrow F\&\pi_2(V,L)$$

where the $*$ indicates that the value returned is not used (and thus again need not be transmitted).

Although experience to date shows fetch-and-add to be the most important member of this class, other fetch-and-ϕ operations will be useful as well in the construction of ultraparallel operating systems. For example, the RP3 provides fetch-and-or as well as fetch-and-and, which together provide a means for atomically setting and clearing bit flags within a word of storage. On a standard architecture without these operations, bit setting and clearing functions require critical sections protected by semaphores.

2.4. The Yet More General Read-Modify-Write Operation

Modern serial machines often include certain atomic read-modify-write memory operations and we wish to consider augmenting the parallel models of computation to include a class of such operations, which we write in the form RMW(V, f), where V is a variable and f, the ''modification'', maps values to values. This operation returns the (old) value of V and replaces it with $f(V)$. A most powerful model would permit the single-cycle execution of all read-modify-write operations. A more feasible model would require that f itself have complexity O(1) on a traditional serial model. Note that loads and stores can be formulated as RMWs: for a load f is the identity; for a store f is a constant and the value returned is ignored. Fetch-and-add, and more generally fetch-and-ϕ, can also be so formulated: F&A(V,e) is RMW(V, f), where f is the increment-by-e function. Simultaneous RMWs are required to satisfy the serialization principle mentioned above. For some classes of RMW operations, simultaneous RMW operations directed at the same memory location (but having different modification maps) can be combined using hardware similar to that described in section 7 for combining fetch-and-adds. More information on this subject can be found in a recent article of Kruskal *et al.* [88].

2.5. The Power of Fetch-and-add

Using the fetch-and-add operation we can perform many important algorithms in a completely parallel manner, i.e. without using critical sections. For example, as indicated above, concurrent executions of F&A(I,1) yield consecutive values that may be used to index an array. If this array is interpreted as a (sequentially stored) queue, the values returned may be used to perform concurrent inserts; analogously F&A(D,1) may be used for concurrent deletes. The complete queue algorithms contain checks for

overflow and underflow, collisions between insert and delete pointers, etc. (see Gottlieb *et al.* [83b])[5]. Forthcoming sections will indicate how such techniques can be used to implement a totally decentralized operating system scheduler. We are unaware of any other completely parallel solutions to this problem. To illustrate the nonserial behavior obtained, 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. The importance of critical section free queue management routines may be seen in the following remark of Deo, *et al.*[80].

> 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.

For another example of the power of fetch-and-add, consider the classical readers-writers problem in which two classes of processes are to share access to a central data structure. One class, the readers, may execute concurrently; whereas the writers demand exclusive access. Although there are many solutions to this problem, only the fetch-and-add based solution given by Gottlieb *et al.* [83b] has the crucial property that during periods of no writer activity, no critical sections are executed[6]. Other highly parallel fetch-and-add-based algorithms appear in Kalos [81], Kruskal [81], Rudolph [82], and Wilson [86].

To understand why bottleneck free algorithms can be constructed from fetch-and-add and not from test-and-set let us first consider the values returned by concurrent executions of F&A(V,1) and then by concurrent executions of T&S(V). With the former primitive, distinct integers are obtained by the issuing tasks, thereby distinguishing each one from all the others. Once distinguished, each task can perform its individual assignment. Since test-and-set is Boolean valued, tasks executing this primitive concurrently are separated into at most two classes. Moreover test-and-set can return the value false to at most one of these tasks. Hence if K tasks were involved, one of these classes has at most one member and thus the other class has at least K-1 members. We see, therefore, that test-and-set is well suited for situations in which it is necessary to single out one task and permit it to perform some special action, i.e. it is well suited for critical sections[7]. Fetch-and-add, in contrast, is well suited for distinguishing tasks, e.g. to have each one access a different element of an array, which is the essence of the concurrently accessible queue algorithms mentioned above. Another important characteristic of fetch-and-add used in the queue algorithms is that the effect of F&A(V,e) on the value of V can be removed by executing F&A(V,-e), i.e. addition (unlike Boolean or) is invertible. Using fetch-and-add with differing values for the increment is the key to other algorithms. For example, to implement readers/writers protocols we permit B concurrent

---

[5]As explained in Gottlieb and Kruskal [81], the replace-add primitive defined in Gottlieb *et al.* [83b] and used in several of our earlier reports is essentially equivalent to the fetch-and-add primitive defined above.

[6]Most other solutions require readers to execute small critical sections to check if a writer is active and indivisibly announce their own presence. The ‘‘eventcount’’ mechanism of Reed and Kanodia [79], although completely parallel in the above sense, detects rather than prevents the simultaneous activity of readers and writers.

[7]It is also well suited for forbidding at most one task from executing a section of code, but this has not proved to be especially useful.

‘‘activities’’, where a reader counts as 1 activity, a writer counts as B, and B is large.

2.6. 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 [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 7.1.4.

The current generation of vector 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]). These supercomputers achieve their full power only on algorithms dominated by 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, *et al.* [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 shared memory model that our simulation studies have shown to be effective for both fluid-type (Korn and Rushfield [83]) and particle tracking calculations (Kalos, *et al.*[81]).

Yet a third alternative model, specifically architectures derived from ‘‘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 applicative languages are indeed attractive for some problems, we do not believe that dataflow architectures are truly general-purpose. For example, it is unclear how a dataflow processor can support simulations of computers, even if the machine to be simulated is itself a dataflow processor. In contrast, we

believe that the Ultracomputer architecture is well suited for a wide class of (time-driven) simulations[8], including performance evaluations of all the machine designs mentioned in this section. The concurrently accessible queues discussed above are used to hold the subcomponents to be simulated during each time step. For another indication of the generality enjoyed by the Ultracomputer architecture, we note that Gottlieb and Schwartz [82] show how a dataflow language may be executed with maximal parallelism on our machine (see also Johnson [81]).

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 is necessary to route messages via intermediate PEs. We subdivide message passing architectures based on whether or not the interconnection topology is visible to the user. In the original ultracomputer design of Schwartz [80] and the Caltech hypercube (Otto and Fox [86]) the topology is visible and the programmer specified the data routing explicitly. By tailoring algorithms to the particular interconnection geometry, one can obtain very high performance. Indeed the very favorable results obtained by Caltech has encouraged significant commercial interest, as exemplified by recent designs such as the Intel, NCUBE , and FPS (T-series) hypercubes. 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 Gottlieb [80a], Gottlieb [80c], Gottlieb and Kruskal [80], and Schwartz [79]). If the topology 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. Parallel Programming

3.1. Levels of Parallel Control

The shared-memory MIMD model can support a number of different styles of programming. Relative efficiency and usefulness of these alternatives are affected by performance issues relating to the underlying system software implementation. We do not address the issue of automatic parallelization of sequential programs and assume instead that programs are explicitly written for a shared-memory MIMD computer.[9]

[8]There is no widely accepted algrotihm for treating event-driven simulations on highly parallel computers. We believe that the ultracomputer will prove to be the best architecture at least for experimenting with such programs.

[9]However, the automatic techniques developed at the University of Illnois (e.g., see Kuck and Padua [79]), Rice University (e.g., see Kennedy [80] and Allen and Kennedy [84]), and IBM (e.g., see Allen *et al.* [88]) will be important for running both existing and new applications.

Much work on concurrent programming has focused on a high-level, block-structured paradigm of parallel process control. Under this model, parallel code is structured in closed-form constructs with implicit synchronization at the end of each parallel block; explicit synchronization and ‘‘fork/join’’ operations (Conway [63] and Dennis and Van Horn [66]) are discouraged or disallowed. The argument is that the resulting parallel code is simpler, clearer, and easier to debug than code with unrestrained and unstructured process creation/destruction and synchronization. Furthermore, this programming style obviates in many cases the need for explicit shared/private declarations. That distinction is instead implicit in the usual static scope rules. Thus, for example, a variable that is visible within a block defining an ‘‘iteration’’ of a parallel loop, but declared in a larger enclosing scope, is taken to be shared during execution of the parallel loop; the scope of a variable declared within the block itself is an individual iteration and hence the variable is private. Finally, automatic optimization of parallel code is facilitated when the parallel structure of a program can be readily detected by the compiler. For example, it would often be possible for an optimizing compiler to implement reliably a finer granularity of control over the cacheability of data areas than could a programmer.

However, the utility and effectiveness of this high-level parallel programming style is not yet demonstrated. Furthermore, the volatile parallelism facilitated by these closed-form parallel constructs will not be needed in all programs. When parallel processes need to synchronize very frequently, the overhead involved in the process creation and termination operations invoked by these constructs will become significant. This overhead can be alleviated to some extent by pre-spawning processes as described below. However, many parallel applications will be more suited to a lower-level programming style. One such approach involves the initial creation of a fixed number of long-lived processes, usually smaller than the number of PEs. Synchronization, scheduling, and memory management become entirely the responsibility of the application programmer. Here the syntactic structure of the program provides no information regarding the dynamic parallelism or the sharing of data.

As we will see in a subsequent section, an attractive synthesis of the two styles may be obtainable. We consider implementing these scheduling and management functions in usermode code that is part of the runtime environment provided by a language compiler. In the ideal case this would afford the advantages of high-level structured parallel programs, while the creation, destruction, synchronization, and management of parallel threads of control are accomplished without operating system overhead. Ramifications of these programming models are discussed below.

3.2. Parallel Constructs

While programming language issues per se are not germane to this paper, for concreteness we present examples of two commonly proposed high-level parallel constructs. Note that programming in the MIMD parallel environment need not be radically different from conventional sequential programming. We consider parallel languages which are variants of conventional procedural languages, augmented only with a shared/private attribute for declared variables and a small number of explicit parallel control constructs.

3.2.1. Control Structures The parallel loop, in which homogeneous iterates are executed in parallel instead of serially, is an obvious parallel extension of the loop construct found in every procedural language (see Gosden [66], Droughon, *et al.* [67], Lundstrom and Barnes [80], and Davies [81]). It may be stated as:

forall *j* **from** *lb* **to** *ub* **step** *inc* **do** *s*

As is the case for a sequential loop, the number of iterations, *i*, performed over the loop body, *s*, is controlled by the variables *lb*, *ub*, and *inc*. However, the *i* iterations of the body are executed in parallel with each evaluation using a different value for *j*. Each iteration is represented by a different process; the operating system arranges that each process is scheduled on a PE for execution. The loop construct provides implicit synchronization; that is, the statements following the **forall** block will be executed (serially) only after all of the parallel iterations have completed executing the loop body *s*.

The parallel compound statement, or parallel block, contains inhomogeneous statements that are to be executed in parallel. It is also a popular structure and has appeared many times (e.g, Dijkstra [68], Brinch-Hansen [73], and the collateral expression of ALGOL 68) and may be expressed as follows:

s_0
parbegin
  $s_1; s_2; ...; s_n$
parend
s_{n+1}

First, the statement s_0 is executed; then statements s_1 through s_n are executed in parallel; finally statement s_{n+1} is executed.

Neither construct contains explicit reference to processors. The PEs themselves are a resource that is only indirectly available to the program. Parallel constructs generate tasks that run on processors. Since the degree of parallelism (number of concurrently active tasks) may be highly volatile, static assignment of processors to programs is not in general desirable (exceptions will be discussed shortly). Rather, tasks are scheduled on processors dynamically. Potentially, all of the tasks generated by a parallel construct could execute simultaneously. Whether this actually occurs is a function of scheduling policy, system load, and various characteristics of the user program and language implementation details.

3.2.2. Shared Variables Support for completely general sharing of data segments between arbitrary subsets of *k* processors could require 2^k separate segments. In practice, sharing patterns will probably exhibit more structure. Using a conventional block-structured language augmented with closed constructs for parallelism in which the usual scope rules apply, a variable can be accessed by any statement within the scope of its declaration. In particular, if that statement is a parallel loop or block, the variable will be shared by all processes executing the statement. It follows that data is accessible within the subtree of processes rooted at the process containing its declaration. This limits the number of visible segments to the height of the process tree. Note that the status of a variable (private or shared) changes through the execution of the code.

As an example, consider the following code fragment (assuming arguments are passed by reference):

```
var X : integer;
parbegin
  var Y : integer;
  f(X,Y);  g(X,Y);
parend;
```

Prior to the **parbegin**, X is private; while the functions f and g are executing, X is shared (by f and g); at the termination of the parallel clause X becomes private again. The variable Y is private. Note also that the status of X and Y (private or shared) in each block of the code is implicitly defined by static scoping rules and can be computed at compile time. This dynamic behavior impacts the memory allocation strategy, paging mechanism, and cache scheme.

It is often necessary that shared variables be accessed atomically. If the size of memory word (the largest unit of information that can be fetched in one access) is increased, then atomic access can be provided to larger structures, but, as discussed in section 7.4, memory access conflicts become more troublesome. This has nontrivial implications for variables larger than a single word, since critical sections may be required to ensure atomic access.

3.2.3. Barrier Synchronization While it is important to minimize the idle processor time caused by synchronization among parallel processes, such synchronization is occasionally necessary. A common form of synchronization occurs when each of the processes executing a parallel code section must wait at a ‘‘barrier’’, or synchronization point, until they have all reached that point (see Jordan [78]). Ultracomputer algorithms for barrier synchronization have been given by Rudolph [82] and Kruskal [81]. Other important types of synchronization are discussed in section 6.3 in the context of the parallel operating system.

3.3. Implementation of Parallel Constructs

Having discussed the benefits of a ‘‘high-level’’ programming style, we now consider in some detail whether that style can be supported effectively. In particular, potential parallelism must not be sacrificed due to the parallel language implementation. We shall see that the *granularity* of parallelizable processes is a crucial barometer of the effectiveness of the implementation.

The basic mechanism provided for creating parallelism is the *spawn* operation, which is used to support the parallel loop and parallel block constructs. Spawn is fundamentally an n-way fork of control, in which n identical subprocesses are created. The subprocesses are made available for scheduling on any available PEs, in a manner to be discussed. We assume here that the ‘‘parent’’ process waits for the termination of its spawned ‘‘children’’, which occurs automatically at the end of the parallel code block. Hence the parent process–and thus the subsequent program statements–are synchronized with the completion of the spawned parallel processes.

The translation into runtime code of a parallel loop, with *n* homogeneous iterates, might make use of the following operating system primitives. First, a *spawn(n)* is executed by the original (parent) process. It stores the value *n* in a shared *children* variable, adds *n* items to the system work queue, and executes a form of *wait* so as to block until resumed subsequently by a terminating child process. The *terminate* function is executed by the spawned child processes at the end of the loop body. *Terminate* executes F&A(*children*,–1); if this drives *children* to zero then the current process is the last terminating child and thus resumes the parent process. An obvious implementation for the central work pool is the fetch-and-add based parallel-access queue that was mentioned earlier.

3.4. Performance Issues in Parallel Control

There are two significant performance criteria by which any implementation of these operating system primitives must be evaluated. First, we wish to avoid algorithms that require time linear (or worse) in the number of spawned processes. Hence it is unacceptable to implement a spawn of *n* processes by a sequence of *n* insert operations on a system process queue. Instead, a spawn operation inserts a *single* item with multiplicity *n* on the process queue, and the PEs deleting such an item complete the creation of the children in parallel. Together with fully parallel memory allocation routines, this will largely prevent the occurrence of diminishing marginal benefits as the degree of parallelism is increased.

However the overhead (in absolute terms) of these operations is also crucial, because it determines the minimum granularity of parallel operations that can be efficiently spawned. Thus, although the basic unit of parallelism provided by the language constructs is the program statement, it is clear that spawning processes that each execute a trivial statement (e.g., one assignment or arithmetic operation) would be inefficient. While careful design of portions of the operating system can reduce this overhead, the facilities described to this point must be considered useful only for relatively large-granularity parallel operations. This limitation can be alleviated in several ways.

Parallel loop iterates of smaller granularity can often be supported with a policy known as *chunking*. When the number of parallel iterates (*n*) is much larger than the number of available PEs (*N*), or the number of instructions in the body of the loop is small compared to the overhead of process creation, then the loop can be transformed into a serial loop consisting of *k* iterations nested inside a parallel loop of *n/k* iterations. Thus the scheduling overhead per iterate is reduced by a factor of *k* invisibly to the programmer, and, if $n/k > N$, the effective parallelism is not diminished. The value of *k* is most appropriately determined at runtime as a function of the number of PEs available and possibly of system load. Nonetheless, this strategy requires regrouping iterates statically (i.e. at the initiation of execution of the parallel loop) into possibly unbalanced parallel tasks. Hence, care must be taken to avoid setting *k* too high and nullifying the load-balancing properties of the ‘‘self-service’’ paradigm discussed below. Kruskal and Weiss [84] have examined the performance of such chunking policies and argue that even very naive schemes can perform acceptably well.

A related scheme, known as *dynamic chunking* or ‘‘self-scheduling’’, uses in effect a parallel queue of iterates built at loop initiation. Each of the *n/k* parallel processes repeatedly deletes and executes

iterates until the queue is empty, thus ensuring effective load-balancing especially when the variance of execution time among the iterates is large. In practice the queue is unnecessary, because the next value in a sequence of integers can be obtained with a single fetch-and-add. Thus dynamic chunking can be accomplished with no greater overhead than in the static chunking outlined above.

Effective granularity of programmed parallel functions can be further reduced by pre-spawning processes. Here the programmer or compiler spawns and suspends a sufficient number of processes in advance of any parallel constructs. The parallel loop or block code then activates these processes by sending a message or a signal, thus ‘‘creating’’ parallel threads of control without the overhead of process creation. In effect we are supporting the parallel constructs with operating system functions moved into usermode code.

Finally, one can pre-spawn non-preemptable processes that busy-wait until needed by parallel constructs. Far finer granularities of parallel functions can thus be programmed with high-level constructs, since virtually all of the overhead of process creation, scheduling, and context switching is eliminated. The cost is in tying up a larger number of PEs than may actually be used during much of the program execution, which is particularly serious if the degree of parallelism is highly volatile.

3.5. Performance Issues in Barrier Synchronization

Synchronization operations may be implemented in two ways: It is always correct for the processor to suspend the current process while waiting for the synchronization to complete, and switch to another process. However, considerable operating system overhead is incurred. The alternative is a busy-waiting synchronization routine that simply loops and tests whether the synchronization condition is satisfied. For short waits, the overhead involved is significantly less for busy-waiting than for process-switching. However, busy-waiting admits the possibility of deadlock when the number of synchronizing processes is greater than the number of available processors (and no preemption is permitted). Even when deadlock cannot occur, it is possible that a number of processors will loop unproductively for extended periods of time, particularly if one or more of the synchronizing processes is preempted or swapped out.

A hybrid implementation, in which each process busy-waits for a short period and then, if the condition is still not satisfied, yields the PE, would avoid rescheduling overhead in those cases where busy-waiting is suitable, and deadlock would be prevented. Synchronization issues have implications for process and job scheduling that are addressed below.

4. Process scheduling

4.1. The ‘‘Self-Service’’ Paradigm

Operating systems for most uniprocessors and some multiprocessors contain a single module that schedules use of the processor(s) by assigning processes as appropriate. While this centralized approach assures favorable load balancing, it introduces a serial bottleneck that will limit overall performance on highly parallel machines. Alleviating this bottleneck by designating two or more schedulers, each

managing a portion of the processors, leads to an interesting tradeoff: if the number of schedulers is small, scheduling bottlenecks arise; if the number is large, effective load balancing suffers.

Another technique used to avoid the bottleneck is stochastic distributed scheduling. Here a work queue is maintained for each PE. Whenever a process is created on any PE, that PE assigns the process on a random basis to one of the work queues. In concert with time-slicing, or when certain constraints (on the variance of process execution times) hold, this mechanism can effectively balance the load, possibly at a cost in scheduling overhead (see Klappholz [82]). It does not permit the constant-time spawning of multiple processes discussed earlier since the assignment of each process must be randomized individually.

The scheduling paradigm adopted for process scheduling (and other resource management functions) in the Ultracomputer is that of a self-service system, in which one maintains a single central queue of ready processes. Each processor accesses this shared queue to obtain processes for execution and to insert newly-spawned processes. This self-service paradigm relies on simultaneous distributed processing of centralized data, and is highly dependent on concurrently-accessible data structures that allow concurrent operations to be performed without serialization. The fetch-and-add based mechanisms described earlier are crucial in implementing these critical-section-free operations, e.g., queue insertion and deletion.

The queue algorithms used are those mentioned earlier enhanced to support three important additional features (see Gottlieb *et al.* [83b] and Wilson [86]):

- *Multiplicity.* A spawn of k processes is implemented by the insertion of an item of multiplicity k, which is ‘‘deleted’’ k times before actually being removed from the queue.
- *Priorities.* Processes may be inserted onto the central ready queue at any one of a (fixed) number of priorities, with the delete operation removing the highest priority item[10].
- *Interior removal.* In order to swap out a process from memory, or to prematurely terminate a process, it is occasionally necessary to ‘‘delete’’ an item from the middle of a queue.

Distributed scheduling from a central ready queue achieves optimal load balancing among the PEs and also facilitates multiprogramming. Unrelated jobs may contribute processes to the ready queue; all will be scheduled as PEs become available. In addition to its usual benefits, multiprogramming can improve throughput by overlapping serial sections and highly parallel sections of different jobs.

The task scheduling scheme described above can be further extended to support a heterogeneous system, containing PEs tailored for specific classes of tasks (e.g. I/O or FFT processors). To ensure that a

[10] In addition to other more obvious functions, process priorities are needed in management of nested spawns. The dynamic process structure of a program in which spawns are nested several levels deep may be depicted as a tree, in which spawned child processes are represented by nodes that are descendants of their parent process node. If the program is executed such that the process tree is traversed breadth-first, then there is a danger that because the processes at each level are spawning more processes before any process may terminate, the capacity of memory or system tables may be overwhelmed by an exponential explosion in instantiated processes. This is avoided by ensuring depth-first traversal of the process tree, which may be achieved by enqueuing the template for creation of spawned children on the ready queue at a priority greater than that of the parent.

task is assigned to an appropriate PE, a distinct ready queue is associated with each task class and only processors specialized for a given class delete entries from the corresponding queue. Other scheduling matters such as *preemption*, *timeslicing*, and *priority aging* can also be easily accommodated within the basic framework described above.

4.2. Scheduling for a Spectrum of Program Styles

As indicated above, when the number of ready processes exceeds the number of processors, the operating system uses a priority based preemptive scheduling algorithm. Although we expect this standard multiprogramming discipline to prove adequate for program development as well as for production runs of many programs, we anticipate that some sophisticated users solving large problems will require finer control over scheduling. Such users are well served by the non-preemptable allocation of a number of processors, which are then assigned to subtasks under program control. In addition to permitting a problem-specific dynamic choice of which subtask to execute next (rather than relying on the operating system's unsophisticated notion of priority), non-preemptable processes are immune from the overhead associated with involuntary context switching. The operating system supports non-preemptable processes with a variant of the *spawn* primitive that inserts a non-preemptable item with multiplicity *n* onto a high priority ready queue, causing the next *n* available PEs to select these processes for execution. Since the operating system will not permit the total number of such processes in the system to exceed the number of PEs, the time delay between the invocation of the first and last instance of the process in question is bounded by the preemption interval.

To illustrate one use of this facility consider an application that requires tight synchronization between processes, i.e., one in which processes must synchronize after executing only a small number of instructions. Although the stochastic nature of the network prevents an exact determination of the rate of progress for an individual process, with preemption removed (and with the Ultracomputer combining network), a group of processes will execute at roughly equal rates with high probability. Thus, providing that the program segments executed by each process between corresponding synchronization points are of comparable length, a programmer using the non-preemptable spawn can profitably synchronize the resulting processes by means of busy waiting loops free of system calls.

In summary, we have considered a spectrum of ‘‘organizational styles’’ of parallel programs and in the next section discuss operating system support for the various styles.

(1) On one end of the scale are jobs that are static in their use of PEs and are subject to real-time constraints. Processes associated with such jobs should not be preempted under any circumstances.

(2) More important is the class of non-real-time applications referred to above that coordinate parallel activities on a fixed group of PEs and are characterized by very frequent internal synchronization. As mentioned, busy-waiting synchronization is appropriate for such jobs. They may be preempted or even swapped out, as long as all of the processes are preempted together.

(3) Jobs displaying dynamic parallelism are better suited to our original model of process scheduling. Although internal synchronization will still be needed occasionally, it can be adequately managed with the hybrid synchronization mechanism proposed at the end of section 3, even in the presence of

chunking. Nonetheless there are reasons to keep related (''sibling'') processes executing concurrently: First, there are algorithms whose performance is improved when parallel processes execute at more or less the same rate, that is, when the execution rate of the slowest-progressing process in the spawned set is maximized. Second, the effectiveness of the processor cache will be improved when successive processes executed on the same PE come from the same job, since they are then likely to reuse cache entries for program code[11].

The structure of the central ready queue is further complicated by this need to recognize groups of sibling processes in swapping and scheduling. However, the original fetch-and-add based notion of bottleneck-free inserts and deletes can be maintained.

5. Operating System Interface

The set of operating system services needed by parallel programs is to a large degree identical to that provided in conventional serial operating systems. We enhance the UNIX kernel interface with a small number of primitives for creation and synchronization of parallel threads of control, and for management of shared and private memory areas. The desire for high performance also has ramifications for other system functions, such as I/O and file system organization, that are not specifically related to MIMD or shared memory systems. No further comments about these last areas will be made in this paper.

There are many different ways of structuring parallel programs, and hence the most important goal of the system interface design is generality. Both process management and memory management offer choices for programming language and application designers that involve complex tradeoffs between ease of program design and debugging, and possible efficiencies to be obtained through low-level coding or lower levels of protection; between efficient accommodation of volatile levels of parallelism and efficient processing of I/O or tight internal synchronization; between turnaround time for a particular job and overall throughput for a set of jobs with varying resource requirements; and so forth. As always in operating system design, the kernel must implement a small set of primitives that provide for the widest possible range of user applications.

Almost all of the facilities described in this section are implemented in a prototype UNIX-based operating system at NYU, which is described later in this paper. However, the kernel interface is very much a work in progress. Far more experience is needed in designing languages and programs for shared-memory multiprocessors before we will fully understand the requirements for the programming model in service of parallel programs. Aspects of UNIX terminology and concepts have been adopted for the present section.

[11] Here we assume that the cache architecture permits retaining cache lines across a context switch.

5.1. Process Management

At the most basic level, each thread of control in a parallel program is embodied in a standard UNIX process. We use the term *job* to refer to the collection of processes executing a single parallel program, normally a subtree rooted at a process created by *fork*. Certain operating system functions such as scheduling, shared memory management, and signaling may recognize jobs as well as individual processes.

5.1.1. System Calls The *spawn* system call has already been introduced. It is a multi-way fork that creates *n* processes in time essentially independent of *n* (unlike an iterated *fork*, which would require time proportional to *n*). Spawned processes are full-blown UNIX processes, and they inherit attributes from the parent much like forked processes, with only minor exceptions[12]. The set of child processes created by a single *spawn* is referred to as a *spawn group*.

Arguments to *spawn* include the multiplicity (*n*), option flags, and, optionally, the location of an array used for reporting of child processes' exceptional termination conditions. Option flags include (1) request for nonpreemptable child processes, essentially a request for *n* PEs, and (2) request for cactus stack processing (to be discussed in Section 5.2.4). The parent process may obtain exit codes from individual subprocesses, or summary counts representing a histogram of the various termination conditions.

The *spawn* call normally transfers control to all processes, much like *fork*, returning the child's ''spawn index'' (a number uniquely chosen from {1,..,*n*}) in each child process and zero in the parent process. Child processes then execute independently until terminated with *exit*.

Mwait (''multiple wait'') is a new system call used by a parent process to await the termination of all spawned children. Again, neither *mwait* nor *exit* involve serial operation, as would be the case using the standard *wait* system call, which would have to be iterated, once for each child created. *Mwait* will also return in the event of an abnormal child termination, with the error status suitably reported. Furthermore, *mwait* can be used to test (without blocking) whether outstanding children remain.

A new signal, SIGPARENT, is automatically sent to all processes spawned by a terminating parent. Unlike the traditional situation with forked processes, there is usually no purpose in allowing continued execution of a spawned orphan process. Experience at NYU has demonstrated the need to assist the programmer in cleaning up orphans after an abnormal termination, especially during debugging of parallel programs.

[12] Spawned processes must however be distinguished from forked processes, for technical reasons that will become apparent.

5.1.2. Low-Overhead Parallel Threads UNIX processes are relatively ‘‘heavy’’ objects. Associated with each is a unique memory management context containing private and shared memory areas, a number of open files, and a substantial number of attributes (userids, current working directory, signal actions, etc.). Process creation requires duplicating much of this state and even context switching can involve considerable overhead, depending on the memory management architecture and other factors. In earlier sections we proposed to reduce the impact of process creation and destruction by pre-spawning processes, and to eliminate the context switching overhead as well by permitting non-preemptable processes. In this last case the program in effect obtains and then schedules a fixed number of PEs; if multiprogramming throughput is not an issue then one might assign all or almost all of the existing PEs to an individual application in this manner.

A natural organization for managing the ‘‘assigned’’ PEs involves user-level threads of control that are scheduled by user code into execution under the UNIX processes, which are fixed on the individual PEs. These threads are known herein as *tasks* to distinguish them from UNIX *processes.* The operating system has no cognizance of these tasks; their creation, destruction, scheduling, synchronization, and resource assignments are accomplished by a layer of software that is part of the user program, most likely included from a standard library, or the language runtime environment. The ‘‘weight’’ of UNIX processes is no longer a concern, since the processes, once spawned, are entirely static. However, such jobs will be unable to respond efficiently to rapidly varying demands for parallelism by the usermode tasks; if the user wishes to dynamically expand or reduce the pool of available PEs then process creation or context switching overhead arises. However, the processor pool can be expanded or reduced slowly, in response to long-term demands or between major phases of a program, without loss of efficiency.

There are further difficulties with this scheme of ‘‘user multitasking’’ in the UNIX process environment. If a task modifies any aspect of the process state, by (for example) opening a file or changing the current working directory, and that task is later executed under a different UNIX process, its environment will invisibly change. File access will fail or affect an incorrect file, the current working directory will be wrong, etc. These problems appear to be solvable within the current framework, although the details are still under investigation. For example, the kernel interface might be extended to allow access to files opened in other processes within the same job, and in general, system calls (e.g. *chdir*) can be intercepted by a layer of software that insures regular system call semantics are maintained for each task.

From the point of view of the operating system kernel, we have considered all processes to be homogeneous (the *job* and *spawn group* defined above are merely aggregates and have no associated attributes or capabilities). In other work in parallel programming environments the concept of *lightweight tasks* implemented in the kernel has proved popular (e.g., Baron *et al.* [85]). Such tasks are scheduled by the operating system but contain almost no private state; rather, they exist within the resource domain of a process or job. Some of the above difficulties would be alleviated if such objects as opened files were maintained on a job level rather than a process level. Other aspects of the process state semantics would change; e.g. the current working directory could no longer be manipulated by an individual program thread. The overall utility of lightweight tasks is not yet known. It is not clear whether they will enable

scheduling of volatile parallel threads with minimal context switch overhead. Further investigation is required in this area.

Memory management issues pertaining to this discussion are considered in the next section.

5.2. User Memory Structure

The operating system must provide one or more segments within a job which permit data to be shared among processes. Here we consider shared memory for storage of data within a parallel program rather than for general inter-process communication. Hence there is no need for memory segments shared among arbitrary unrelated processes, and there may be no need for dynamic creation of shared memory segments other than in the course of program or process initiation. As indicated above when discussing shared variables in user programs, full generality together with full protection would require a very large number of shared segments. In a pure global shared memory environment, however, a simpler, more structured approach appears adequate and natural. Each shared data area is accessible over a subtree of processes; it is created on behalf of the parent process and inherited by all spawned descendants. Thus, the number of shared segments visible to an executing process is bounded by the spawn nesting level. When sharable local memory is present, as in the RP3, further mechanisms for management of shared segments will be required.

The process image strongly resembles that of traditional UNIX processes. Logical memory segments for shared program text, private data, and program stack remain, though in some cases transformed. We augment these with one or more intra-job shared data segments.

5.2.1. Object Files Traditionally the text and data segments are initialized by the *exec* system call from the text, data, and bss (uninitialized data) segments of an object (*a.out*) file. In the parallel environment more object file segments are required. The shared data segment is created from shared data and shared bss segments in the *a.out* file. Furthermore, in architectures supporting local PE memory, we may need the capability to specify at compile time program components to be loaded into the local memory. This gives rise to an additional three object file segments for local text, local data, and local bss.

In our prototype operating system we have adopted the Common Object File Format (COFF) from AT&T System V UNIX, although in most other respects our system is based on Version 7 (and in the future 4.3 BSD) UNIX. COFF provides for varying numbers and types of segments in the *a.out* file, and permits the needed flexibility.

5.2.2. Shared Data Segment This segment is created at program initiation (by *exec*), although it may be of zero length. It is inherited by all descendant processes. The segment may be expanded at any time through the new *shbrk* system call, which is usually used via a library parallel memory allocator known as *shmalloc*. The sharing of this data segment is managed much like the traditional UNIX shared text segment, except that it is read-write and exists only within a single job.

Because the shared data segment includes read-write variables, accesses are in general not cacheable. However, there are various circumstances in which certain variables are used, either temporarily or permanently, in a private (accessed by only one process) or read-only manner. A process may dynamically specify the cacheability of such variables. Means will be also be provided for flushing and invalidating the cache as necessary. See section 7.4 for more information.

5.2.3. Private Data Segment One such segment is created for each spawned or forked process and is controlled via the standard *brk*/*sbrk* system call. As in serial UNIX, private data segments are isolated by memory mapping hardware so that even in case of user program error there is no possibility of a private data segment owned by one process being modified by another process. Accesses to the private data segment are always cacheable.

In standard *fork* semantics, data in the private data segment are copied into the new private segment created for a new process. When applied to spawned processes, this policy dictates the following programming language semantics: Private variables replicated for a child process (e.g. an iterate of a parallel loop) are initialized to the current value of the corresponding variable in the parent's private space. It is entirely possible that a programming language will require different behavior, e.g., reinitialization according to an initializer or default value instead of a value propagated from the parent. The *spawn* system call may thus provide an option requesting reinitialization instead of copying of the private segment.

We now consider the impact of user multitasking on the private data segment. An immediate obstacle arises. If the private variables of usermode tasks are allocated in the private data segment, then each time a task moves from one process to another its private variables will have to be copied, or at least remapped, from one private data segment to another. In either case, sufficient overhead is introduced into the usermode context switch to abrogate much of the advantage of this type of program organization. A solution is to place the task private variables in cacheable areas of the shared data segment, so that they are accessible from any process, and rely on the compiler and user code to (1) protect these private subareas from improper access, and (2) issue the appropriate cache flush or invalidate during the user level task switch. Through avoiding use of the private data segment, most of the task switch overhead has been eliminated. Experience will be needed to determine the dangers of private data areas that are not protected or isolated by the mapping hardware. Depending on the programming language and compiler, it is possible that debugging of parallel programs will be more difficult than otherwise. Similar issues regarding the private data segment arise in other situations that involve pre-spawning.

5.2.4. Program Stack Segment This segment involves some of the same issues as the private data segment. In standard UNIX, the stack is merely a second private data segment, distinguished by the fact that it expands and shrinks automatically. The simplest policy in the parallel environment is to replicate the entire stack segment on *spawn* as is done for *fork*. The stack is thus entirely private and cacheable.

However, only the stack frames created since the last spawn need to be private. Furthermore, sharing the remainder of the stack will permit realization of the scope-based paradigm for variable sharability

in structured parallel code that was discussed in Section 3. When parallel constructs are nested in a block-structured language, the automatic variables declared at each level are allocated in successive stack frames. A natural implementation of the scope rules is to arrange that a parent's stack frame be shared by its child processes, and, in fact, by all those processes constituting the subtree rooted at this parent process. The resulting structure is known as a *saguaro* or *cactus* stack (Hauck and Dent [68]). Private stack frames for active processes are linked to the parent's stack frame. There is no serial overhead in creating or destroying these private frames since concurrent memory requests can be processed in parallel. An option flag in the *spawn* system call is used to cause cactus stack (sharing of existing stack) rather than private stack (copying of entire stack) processing[13].

When user multitasking or other forms of pre-spawning are involved, the stack segment presents the same problems as the private data segment and once again a solution, analogously, is to allocate space for all required program stacks in the shared data segment. Usermode code then manipulates the stack pointer register in the course of scheduling tasks, implementing logically private stacks inside the physically shared area. Using cacheable and noncacheable areas as appropriate, a cactus stack can be implemented in this manner. We may now conclude that the only user memory segment needed for such programs is the shared data segment. Private data and stack segments need not even be created. Again, there are potential dangers resulting from the lack of inter-task hardware storage protection.

5.2.5. Local Memory In architectures that support local PE memory as well as global memory, one needs extra logical segments to provide local versions of the text, data, and stack. The RP3 further allows access by each PE to the local memory of every other PE. This is used for message passing and restricted cases of data sharing.

The required kernel interface facilities for support of local memory features are not yet well understood. Explicit allocation of private local memory segments may be provided. Furthermore, the program text and possibly the stack may be ‘‘cached’’ in local memory invisibly to the user. When a local memory is insufficient to service all allocation requests, it may be possible to use the global memory as a backing store, managed with virtual memory techniques. Several additional approaches for utilizing local memory are also under investigation.

5.3. Usermode Synchronization

Both busy-waiting and process-switching synchronization occur among parallel processes or tasks within a user program. Kernel primitives are provided to support both forms.

[13] In order to properly manage the cactus stack, a new activation record must be created for each child process. Our implementation, influenced by the need to support existing languages, satisfies this requirement by having *spawn* invoke each child rather than just return control to it. A visible effect of this implementation is that, when the cactus stack option is selected, *spawn* must be passed an additional parameter, the address of a subroutine or code block to be executed by the children. The semantics of spawn are further modified so as to return control to the parent only after all the children have terminated.

5.3.1. Busy-waiting synchronization The operating system is for the most part not involved in busy-waiting synchronization code in user programs. No facilities beyond access to shared variables (and perhaps fetch-and-add) would appear to be required to implement such routines. However, two difficulties arise.

(1) In programs with signal-handling routines, it may occur that a process holding a lock is interrupted by a signal whose handler requires the same lock. Since the signal handler operates within the same process, deadlock will result.

(2) When busy-waiting locks are used by a collection of preemptable processes, it is possible that the process holding a lock will be preempted and perhaps swapped out while the rest of the processes loop unproductively for a substantial period of time. Thus some programs occasionally need to be guaranteed essentially uninterrupted execution for a very short period of time in order to avoid a severe loss of performance.

A new kernel interface feature is used to avoid these problems. The user program can request temporary suspension of signal delivery and/or preemption during busy-wait critical sections. The latter is only a hint that the kernel may ignore, since it does not affect correctness. It is unreasonable to implement these functions as system calls, which would increase manyfold the expense of a busy-wait synchronization. Instead, the user program sets these temporary modes through specially-designated communication flag variables, allocated in user memory. The kernel is informed of the location of these flag words with a system call (so as to avoid ‘‘magic addresses’’ encoded into the kernel), and checks the flags at appropriate times. Thus negligible overhead is added to the user’s synchronization routines.

5.3.2. Process-switching synchronization Two new system calls, *block* and *unblock*, are added for reliable suspension and reactivation of processes in process-switching usermode synchronization routines. The kernel does not provide semaphores or other coordination facilities directly, but merely manages process status.

The kernel maintains a per-process *pending unblock* flag, which is used to avoid races and deadlock in the user program. The *block* system call atomically tests the *pending unblock* flag, and, if set, clears the flag and returns immediately; otherwise, it suspends the process by entering a blocked state (distinguished from the state set by *pause* or *sigpause*). The user may prevent signals from prematurely awakening the process, as in *sigpause*. The *unblock* system call atomically tests whether the process is suspended by a *block*, and, if so, makes it ready; otherwise, it sets the *pending unblock* flag for the process. The *block* and *unblock* primitives are used in the obvious manner with the proviso that the caller of *block* must be prepared for premature returns. This is because the *pending unblock* flag may be leftover from a previous coordination operation. Code invoking *block* must loop so as to re-*block* if the awaited condition is not yet satisfied.

6. Building the Parallel Operating System

Since programs written for an Ultracomputer-like machine will generate hundreds or thousands of concurrent activities, we will encounter a correspondingly high level of simultaneous requests for operating system services. Serial processing of these requests will generate unacceptable bottlenecks on a large machine. Therefore, the kernel must itself be a highly parallel bottleneck-free program.

We have already outlined an implementation of processor scheduling satisfying these constraints. The synchronization primitives and data structures described below in the context of the operating system are equally applicable, with minor implementation differences, in user applications. These algorithms have been implemented in an experimental operating system running on a prototype (8-processor) Ultracomputer. Based on UNIX Version 7, the experimental system is symmetric (i.e., there is no master-slave relationship) as well as parallel. As described earlier, the system incorporates facilities for parallel applications programs. Work has begun on a 4.3 BSD-based follow-on system. The experimental hardware and software are further described in section 10.

6.1. Data Structures

Our operating system functions make heavy use of centrally stored concurrently accessible data structures made possible by the fact that simultaneous references to the same memory location can be accomplished in the time required for one reference. Here we consider a few structures that have proven useful.

We have avoided pure linked lists, since we know of no bottleneck-free algorithm for deleting items in a linked list. The desirable characteristics of linked lists must be found in other structures. As will be seen, many of these structures use linked lists as subcomponents.

6.1.1. Queues Queues similar to those employed for process scheduling are used by synchronization primitives: The set of processes waiting for a lock or an event are held on such a queue.

The queues used by the operating system are somewhat different from the simple array implementation discussed earlier. We obtain queues of unbounded size by associating a linked list, protected by a semaphore, with each element of the array. An insertion at array element *j* appends its item to the list associated with element *j*. The maximum concurrency supported by this structure equals the number of lists, i.e., the size of the array (see Rudolph [82]). A variation of this queue structure in which FIFO ordering is relaxed is also frequently used, e.g., to manage a pool of free items to be allocated.

6.1.2. Hash Tables A similar structure results when hash tables are used to access indexed (dictionary) information. The size of the hash table (number of buckets) is set according to the desired maximum concurrency, usually the number of PEs times a small factor. The buckets are linked lists, protected by readers-writers locks, so that when no updates are active, items are accessed without serialization. An example is given below.

6.2. Memory Allocation The creation of a new process requires allocation of space for its u-block[14] as well as its private data and stack segments, if any. As with process management, we adopt a self-service mechanism. A number of parallel algorithms for memory management have been designed (see Wilson [86]), including (non-demand) paging, two variants of the Buddy System (Knuth [68]), and a boundary tag method (Knuth [68]). All are parallel analogs of serial algorithms. All, except for one of the Buddy System variants, maintain queues (whereas their serial analogs keep lists) of free memory blocks. Insertions, deletions, and accesses to blocks within a concurrently-accessible queue are at the core of these algorithms; as a consequence we obtain critical-section-free memory allocation.

6.3. Coordination Primitives In order to permit tasks to cooperate, it is often necessary to coordinate their accesses to shared data structures. An ideal situation for parallel execution occurs when completely asynchronous behavior is permitted, as in some ‘‘chaotic’’ application programs (see Chazan and Miranker [69]). Unfortunately, however, it is sometimes necessary to coordinate processes’ accesses to shared data structures. In these cases one must be careful to permit as much parallelism as possible. Here we discuss three mechanisms for process coordination that we have used successfully in our kernel, namely *counting semaphores*, *readers/writers* locks, and *events*. When designing such mechanisms, one must specify whether a processor denied permission should (busy) wait, or suspend execution of the current process and switch to another.

6.3.1. Busy-Waiting Synchronization Despite the potential for waste in busy-waiting, there are several reasons for using it, including potentially low overhead and applicability to situations where process-switching is inappropriate[15]. We have used busy-waiting counting semaphores and readers/writers synchronization extensively and have described algorithms for them before (see Gottlieb *et al.* [83b]).

Semaphores are often used to serialize access to a small portion of a larger concurrently accessible data structure; for example, the individual linked lists used to implement queues of unrestricted size.

Readers/writers synchronization is used naturally in cases where exclusive access is required only infrequently. We also support upgrading a read lock to a write lock and downgrading a write lock to a read lock, and have used the resulting protocols to implement search structures that must support the operation: ‘‘search for an item, and insert it if not found’’. The inode table in our UNIX kernel, for example, is implemented in this manner. Such a structure uses linked lists accessed through a hash table, as described in Section 6.1.2. By performing the search with a read lock, serialization is avoided in many instances including the important case in which many processes search for the very same inode (e.g., the root inode). Only if the inode is not found is it necessary to upgrade to a write lock. The upgrade

[14]The ‘‘user block’’ is a per process swappable memory segment used by the UNIX operating system to contain processes specific information that need not always be memory resident.

[15]E.g., in the implementation of process-switching synchronization mechanisms themselves.

operation may fail in which case the process goes back to search again (while the one process that succeeds performs the insert). The technique of hashing to get good average case performance and using special techniques (here readers-writers code) for concurrent assesses to the same object is quite natural. A hardware analogue is used in our processor to memory interconnection network, see section 7.1.5.2 below.

6.3.2. Process-Switching Synchronization Synchronization without busy-waiting is commonly used in multiprogramming systems since it permits a processor to continue performing useful work when the progress of the current process is logically blocked. As in other multiprocessor UNIX implementations (e.g. Bach and Buroff [84], Felton *et al.* [84]), we have replaced the internal kernel *sleep* and *wakeup* mechanisms. For each of the new mechanisms described in this section, when a process must block, the PE places it on a queue associated with the condition to be satisfied, and executes the next process from the ready queue. When the condition is eventually satisfied, the blocked process is moved from the waiting queue to the ready queue. Unlike other implementations, we avoid critical sections in many cases by using fetch-and-add plus concurrently-accessible queues.

The best examples of non-busy-waiting synchronization come from the area of I/O processing, which takes on special significance for the Ultracomputer, because large numbers of processes can simultaneously perform related I/O operations. For example, searching important file system directories would be a bottleneck if serialized. Since a group of processes reading such a directory would likely all attempt to read the same disk block; serialization would be devastating.

At a low level, physical I/O devices require serialization; this is easily provided by semaphores. Apparent parallelism can be achieved via in-memory buffer caching. Once a disk block is copied into a memory buffer, it may be concurrently accessed for reading; readers/writers synchronization is appropriate in this situation.

Process-switching synchronization is also used to implement *events*, which are often associated with external occurrences, such as the completion of an I/O operation. At such a time, the event is *signaled*, remaining in this state until *reset*. Additional signals (before the reset) have no effect. Since the event might never happen (consider input from a user's terminal) and we need to support the abortion of processes via external signals, it must be possible to terminate a process blocked on an event. We have developed more general procedures of *premature unblocking* for all of the non-busy-waiting synchronization primitives described above by using the *interior removal* primitive previously mentioned in Section 4.1.

There is a special difficulty in designing bottleneck-free algorithms for readers/writers and events, because the most natural implementation would require a single process to completely empty a queue. For example, when an event is signaled, all processes waiting for it must be awakened. One solution to this problem is to move the wait queue as a single object onto the system ready queue, where it will be treated in much the same way as an item with multiplicity. Our current implementation lets newly awakened processes ''help out'' by waking up other processes. This latter approach is less complex than the

‘‘queue of queues’’ method, but requires logarithmic instead of constant time to execute.

7. Machine Design

In this section we sketch the design of the NYU Ultracomputer, a machine that appears to the user as a realization of the computational model presented in section 2, and we justify our design decisions. As indicated previously, no machine can provide the single-cycle access to shared memory postulated in the model; our design approximates this ideal by using a message switching network with the geometry of the Omega-network of Lawrie [75][16] 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 of our model 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.

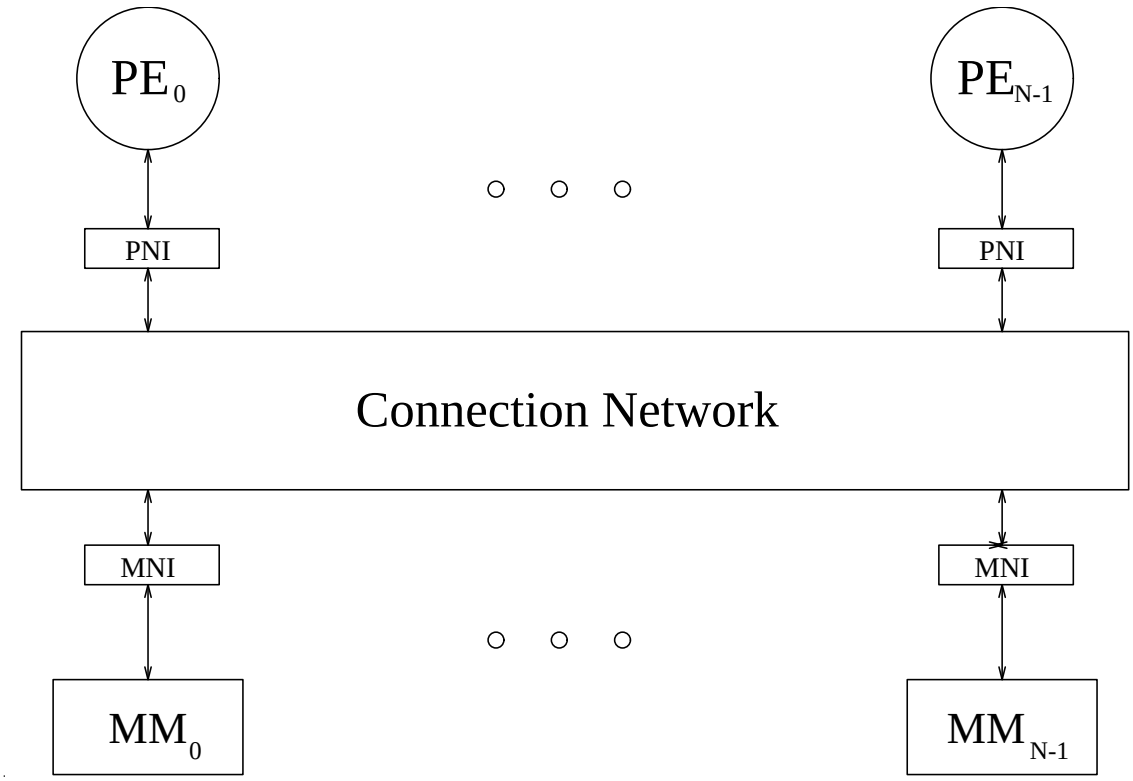

Figure 1. Block diagram.

[16]Note that this network has the same topology as the rectangular SW banyan network of Goke and Lipovsky [73].

After reviewing routing in an omega network, we show that an analogous network composed of enhanced switches can combine simultaneous requests (including Fetch-and-adds) directed at the same memory location, and thus can satisfy any number of such requests in the time required for just one. We then describe the VLSI design of the enhanced switches and comment on other network issues. The network interfaces, the processors, and the memory hierarchy (including caching and paging considerations) are discussed next and the section concludes with a presentation of our work on input/output, two new message passing primitives *reflection* and *refraction*, and machine packaging.

7.1. 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.

- Bandwidth linear in N, the number of PEs.
- Latency, i.e. memory access time, logarithmic in N.
- Only O(N log N) identical components.
- Routing decisions local to each switch; thus routing is not a serial bottleneck and is efficient for short messages.
- 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.

7.1.1. 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 $x_D...x_1$; 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. A message is transmitted from PE($p_D...p_1$) to MM($m_D...m_1$) by using output port m_j when leaving the stage j switch. Similarly, to travel from MM($m_D...m_1$) to PE($p_D...p_1$) a message uses output port p_j 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 2×2 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 2×2 switches, all the results generalize to larger switches, which are considered in section 8.

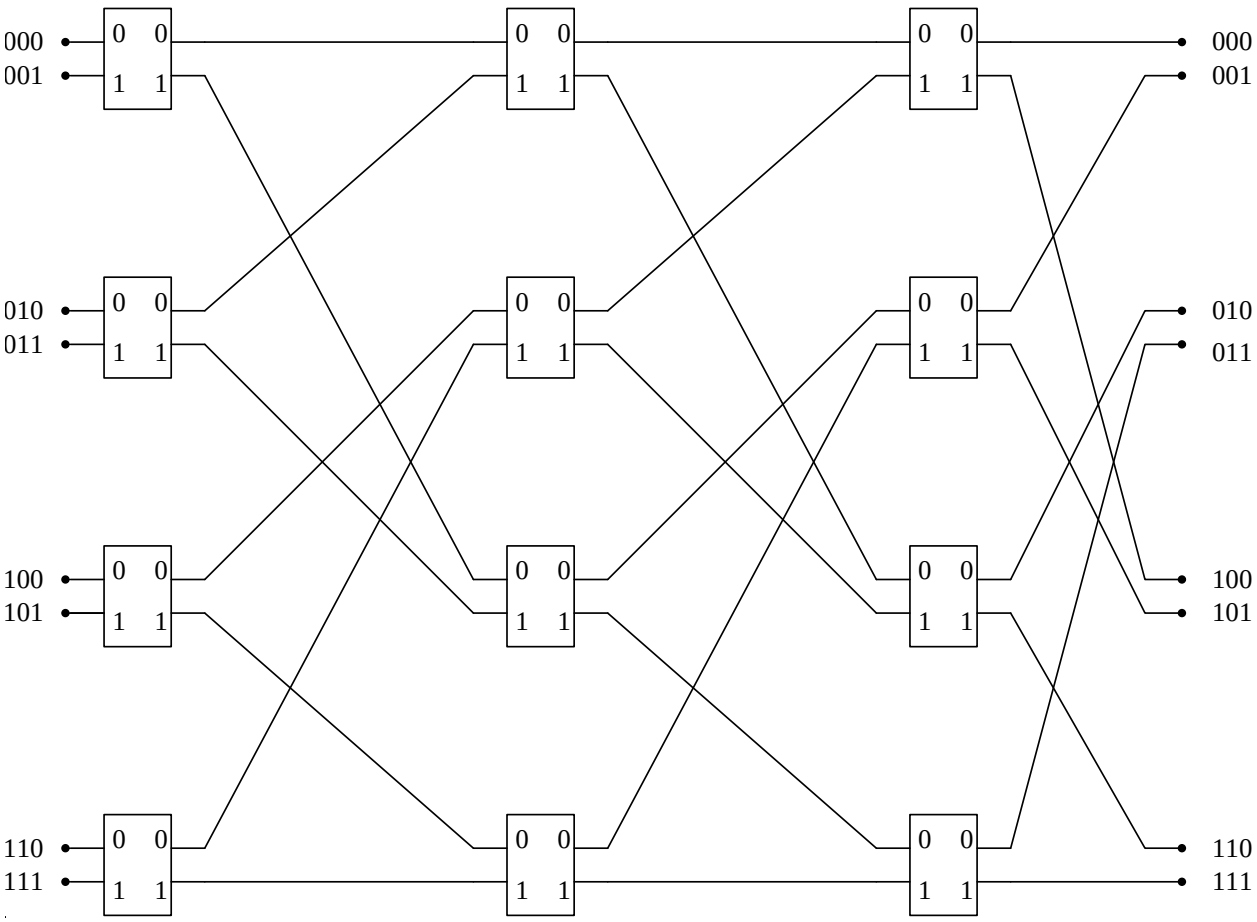

Figure 2. Omega-network (N = 8).

7.1.2. 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 8 for an analysis of network bandwidth):

- 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).)
- 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.)
- 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 [83].)

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 m^j and then replace this bit with the PE number bit p^j, 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.

7.1.2.1. Combining Loads and Stores By executing synthetic data (and small portions of real applications) on a network simulator, Pfister and Norton [85] found that frequent access to the same memory location, a so called ‘‘hot-spot’’, is disastrous unless the network supports combining of such requests. 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 Gottlieb *et al.* [83b] and Klappholz [81]).

- Load-Load: Forward one of the two (identical) loads and satisfy each by returning the value obtained from memory.
- Load-Store: Forward the store and, when the store acknowledgement arrives at the switch, return the stored value to satisfy the load.
- Store-Store: Forward either store and return the acknowledgement to both.

In order to support combining, there cannot be multiple identical requests active in the network at one time. We have adopted a simple policy of not allowing a single processor to have multiple requests outstanding to the same memory location[17].

Combining requests reduces communication traffic and thus decreases the lengths of the switch queues, 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.

7.1.3. Implementing Fetch-and-add By including adders in the MNIs, 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),

[17]An alternative is to send some tag bits that distinguish otherwise identical requests. In this case one must be concerned that the method given for combining loads with stores always serializes the store before the load. If the load was issued first and both came from the same processor, sequential consistency can be violated.

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.

7.1.3.1. Combining Fetch-and-adds 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 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 7.1.4).

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

- Fetch-and-add–Fetch-and-add: As described above, a combined request is transmitted and the result is used to satisfy both fetch-and-adds.
- Fetch-and-add–Load: Treat Load(X) as F&A(X,0).
- Fetch-and-add–Store: When F&A(X,e) meets Store(X,f), transmit Store(X,e+f) and, upon receiving the acknowledgement, satisfy the fetch-and-add by returning f.

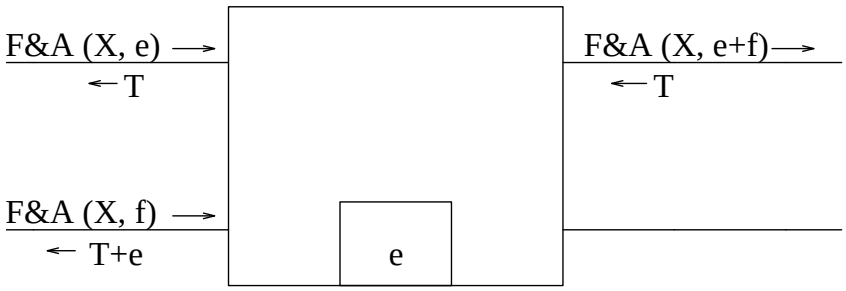

Figure 3. Combining Fetch-and-adds.

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 (Gottlieb and Kruskal [81]).

7.1.4. Network Switches We now proceed to describe the VLSI design of the network switches; additional information can be found in Dickey *et al.* [85] and Dickey *et al.* [86b]. The goals of the switch design are the following:

- Distinct data paths do not interfere with each other. Therefore, a new message can be accepted at each input port provided queues are not full. In addition, a message destined to leave at some output port will not be prevented from doing so by a message routed to a different output port.
- A packet entering a switch with empty queues when no other message is destined for the same output port leaves the switch at the next cycle.
- The capability to combine memory requests should not unduly slow the processing of requests that are not to be combined.
- Flow control information is computed and transmitted in parallel with messages.

The major constraint on network performance is the delay inherent in off-chip communication between VLSI switching nodes, rather than the rate at which information can be processed within each node. Therefore, significant amounts of logic can be added to each node with advantage, when that logic would help avoid global signaling and reduce bottlenecks within the network.

We now detail an individual network switch, shown in figure 4, which is essentially a 2×2 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 $FromPE_i$, where i=0,1, and output ports, called $ToPE_i$. Similarly, the MM side communicates with the MMs via ports $FromMM_i$ and $ToMM_i$. (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). The two-input queues shown in figure 4 are each implemented as two one-input queues plus an output arbiter.

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 combining queue, we search the requests already in this queue using as key the function, MM number, and internal address from

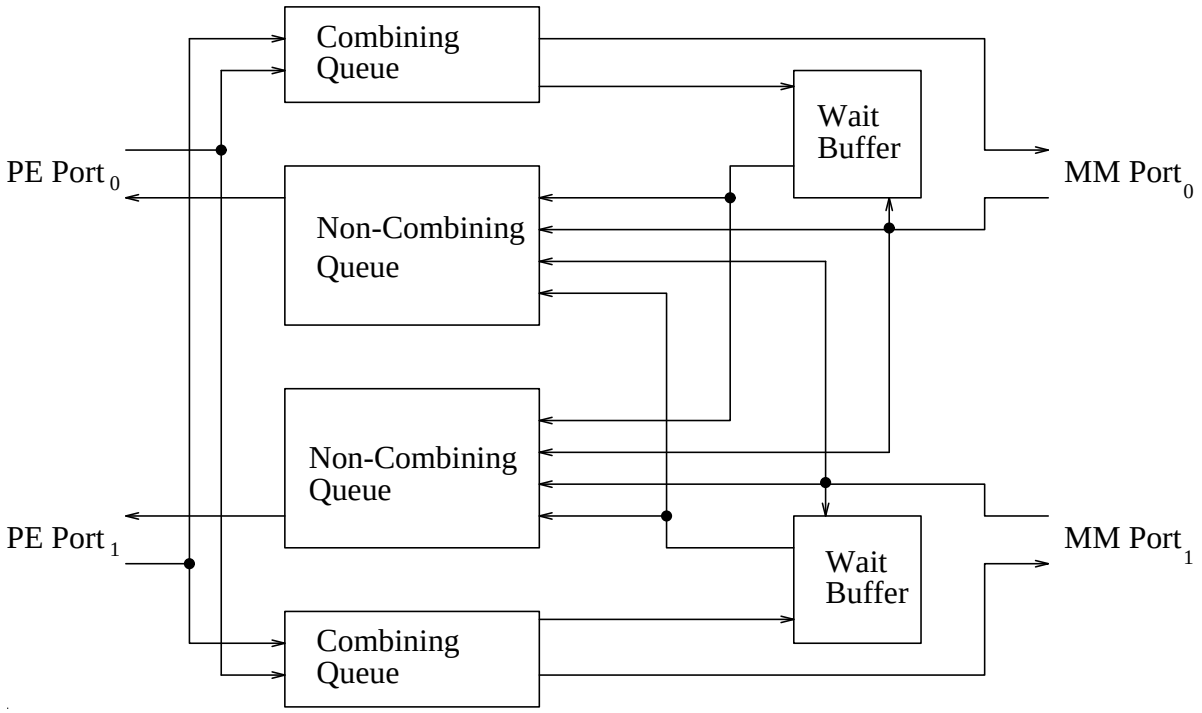

Figure 4. Block Diagram of Network Switch

R_{new}[18]. 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 will use two Wait Buffers (one associated with each ToMM queue) since access to

[18]The design of the ToMM queue, permitting this search and subsequent actions to be performed with minimal delay, is detailed below.

a single wait buffer would be 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.

- ToMM-queue: Entries are inserted and deleted in a queue-like fashion, and matching entries may be combined.
- ToPE-queue: Entries may be inserted and deleted in a queue-like fashion.
- 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. Studies by Lee *et al.* [86] indicate that for thousands of processors three-way combining is required to prevent the ‘‘hot-spot’’ delays mentioned above. 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. The pipelined implementation of the ToMM queue described below does not support multiple combining. A new VLSI design that can combine up to three requests has recently been discovered and a detailed design is under development. Since the new design is not always able to combine three requests, the results of Lee *et al.* [86] cannot be applied directly and a simulation study is beginning.

7.1.4.1. Packaging the Switch For packaging reasons, each switch is divided into a forward path component (FPC), consisting of the two combining queues, and a return path component (RPC) consisting of the wait buffers and non-combining queues. Data forwarded to a wait buffer from a combining queue are transmitted from the FPC to the RPC via ports called wait buffer output ports (WBOPs) and wait buffer input ports (WBIPs) on the FPC and RPC, respectively.

The number of chips required to implement each switching node is determined mostly by the high pin count required at each node, rather than the silicon area of the switching logic. Therefore, messages must be split into multiple packets and one of two methods can be used to transmit these packets through the network. The first is a bit-sliced implementation in which different components are handling different packets of one message (transmission of messages is ‘‘space-multiplexed’’). Or the transmission of successive packets of a message can be time-multiplexed to the same component.

Space-multiplexing provides a higher bandwidth than time-multiplexing at the expense of more components. However, a large amount of ‘‘horizontal’’ communication and coordination must then take place between the different components of a switch, as routing and combining decisions have a global effect. This further increases both the complexity of such implementation and the switch cycle time. For MOS technologies, the off-chip delays impose an especially high overhead.

Several cycles are required to transmit each message if time-multiplexing is used. However, the internal logic of the switch can be pipelined so that messages can be handled on a per-packet basis and do not have to be assembled at each switch. Thus there can be as little as one cycle delay per switch for each request when queues are empty and hence time-multiplexing contributes an additive term to the delay rather than a multiplicative factor. However, queuing delays increase quadratically with the multiplexing factor, so that the performance of the network under heavy load may be seriously impaired (Kruskal and Snir [83]). In the current design we have chosen to use time-multiplexing, so that each message is divided into one packet containing the path descriptor, address and opcode, plus one or more data packets[19].

7.1.4.2. Flow Control The protocol used to transmit messages between switches is a message-level rather than packet-level protocol. That is, packet transmission cannot be halted in the middle of a message. A switch will accept a new message only if the available space in its queues guarantees that it will be able to receive the entire message.

The construction of the systolic queues requires that there be an even number of packets per message and that switches distinguish even and odd cycles. At initialization, the parity of the cycle for the FPC of a given switch is defined to be the the parity of the stage to which the switch belongs, so that cycles that are even for an FPC are odd for the FPCs to which it is connected, and vice versa. Reception of messages starts only at even cycles while transmission of messages starts only at odd cycles. Since the FPC produces messages for the wait buffer at the same time as for successor FPCs, wait buffer transmissions leave wait buffer output ports beginning at odd cycles. The current RPC design requires that messages arrive at wait buffer input ports and FromMM input ports at cycles of the same parity, which for consistency with the FPC are referred to as even cycles. Hence within a switch, the RPC cycle parity of an RPC is defined to be the opposite of the FPC cycle parity[20].

Each port consists of data bits and two protocol bits: a data valid bit (DV) traveling in the same direction as the data and a data accept bit (DA) traveling in the reverse direction. In addition, input ports receive a routing (RO) bit whose value at the first cycle of a message transmission indicates to which

[19] At the expense of a severe increase in complexity, the address can also be transmitted in more than one packet as described in Snir and Solworth [84].

[20] This implies that memory accesses must take an even number of (switch) cycles. Alternatively, an RPC design that accepted messages at the wait buffer and FromMM ports during cycles of opposite parity would require memory accesses to take an odd number of cycles.

output port the message is destined. The RO bit accompanying a packet entering a PE port at the i-th stage of the network is the i-th most significant bit of the MM number; at an MM port and WBIP it is the i-th most significant bit of the PE number.

The two protocol bits, in conjunction with the RO bit, regulate the transmission of messages through the network. A sender asserts DV when it wishes to initiate a message transmission. Independently, a receiver asserts DA when it is able to accept a new message. A message transfer starts only if both DV and DA are asserted and the cycle parity is correct. Since these control signals are ignored during cycles when a message transfer cannot be started, they can be set ahead of time to overlap data transfer and flow control operations. Note that this is not strictly speaking a handshaking protocol: DA is not an answer to DV, nor an acknowledgment, but is issued independently and simultaneously. The sender is transmitting the data on the data lines whenever DV is asserted. If it receives DA, it assumes the data has been accepted and proceeds with the next packet. No provision for retry is necessary.

7.1.4.3. The Forward Path Component The FPC routes requests from PEs to MMs and consists of four one-input combining ToMM queues. As illustrated in Figure 5, each ToMM queue is an enhancement of the VLSI systolic queue of Guibas and Liang [82]. We first describe the queue-like behavior of

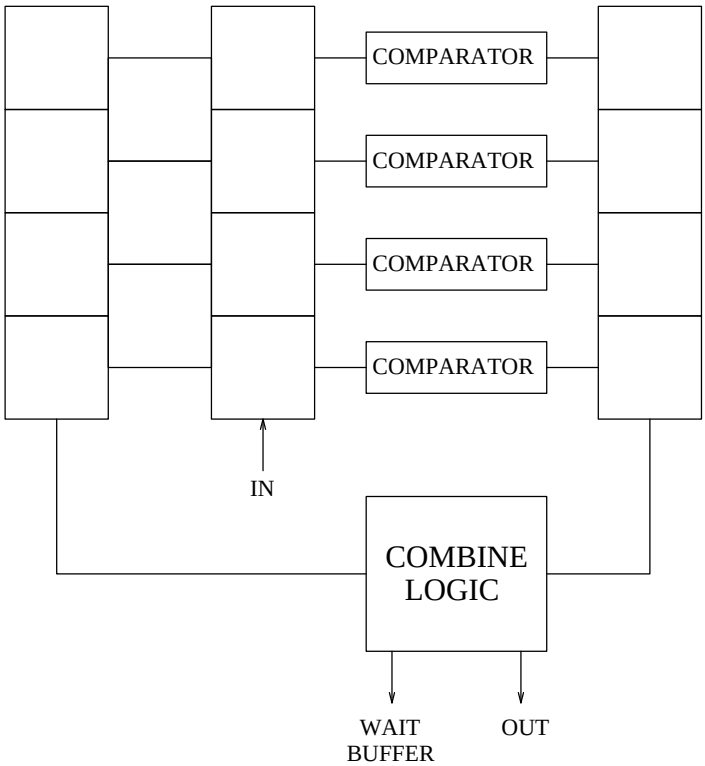

Figure 5. Systolic ToMM queue.

this structure and then explain how the necessary combining 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 giving the enhancements needed for combining, 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[21]; and 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[22]. 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.

7.1.4.4. Return Path Component The RPC routes responses from MMs to the requesting PEs. When a response to a request previously combined by the FPC is detected, the RPC will generate an additional response for the other requesting PE. Each RPC has two MM (input) ports (IPs) and two PE (output) ports (OPs). In addition, two wait buffer input ports receive information from the FPC.

An RPC contains two wait buffers, WB_0 and WB_1, one associated with each MM port and two four-input non-combining queues, which are implemented as eight single-input non-combining queues, Q_{ijk}, $0 \le i,j,k \le 1$. (To enable each input to accept a packet every cycle and to prevent a blockage of one output from interfering with the other output, one queue is required for each input/output pair, where inputs include both wait buffer and MM input ports.) Queue Q_{0ij} is fed from IP_i and writes on OP_j; queue Q_{1ij} is fed from WB_i and writes on OP_j.

A message received on IP_i starting at cycle $2t$ with routing bit set to j is sent to Q_{0ij} and also to WB_j, where its address packet is compared with the messages currently in the wait buffer. If a match is found, the wait buffer asserts its *match* line during cycle $2t+1$, but defers sending its generated response to Q_{1ij} until cycle $2t+2$ so that queues Q_{0ij} and Q_{1ij} receive the first packets of their messages at cycles

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

[22] Actually, an item is matched against half of the entries moving down the right column. Since we packetize messages (see below) we can arrange for each item to be matched against its corresponding item in each request moving down the right column (this is particularly easy if each message consists of an even number of packets). If, however, an entire request is contained in one packet, then one needs either twice as many comparators or two cycles for each motion.

of the same parity.

The DA signal can only be asserted at IP_i if there is at least one empty slot in each of the two queues Q_{0i0} and Q_{0i1}. In addition, there must be sufficient room in queues Q_{1i0} and Q_{1i1} for messages from WB_i corresponding to both the last message received at IP_i and the current message. Therefore, DA also cannot be asserted unless WB_i does not assert *match* and there is one empty slot in each queue Q_{1ix} or WB_i asserts *match* and there are two empty message slots in each[23] of these queues.

To arbitrate between the four queues Q_{xyk} that can send data to OP_k, the RPC keeps track of when each queue has last sent a message. Of the queues that are non-empty, the one having sent least recently is selected.

The DA signal is asserted at a wait buffer input port if the wait buffer will have an available slot to receive a message. As will be seen below, a slot in the wait buffer is capable of simultaneously receiving and transmitting a message. Therefore, DA will be asserted on $WBIP_i$ if WB_i either does not assert the *full* signal or asserts the *match* signal.

7.1.4.4.1. Wait Buffer The wait buffer is an associative memory that stores information sent by the FPC when combining two F&A's into a single request. The wait buffer inspects all responses from MMs and searches for a response to a request previously combined by the FPC. When it finds a response to such a request, it generates a second response and deletes the request from its memory.

The structure of a wait buffer[24] (WB) is shown in Figure 6. A typical message slot is shown in the solid black box and consists of two registers (called Areg and Breg), compare logic, and a controller. Each register contains the data bits, a data valid (DV) bit, and, for the first packet of each message, a routing (RO) bit. The registers are connected in a loop of length two, and shift at each cycle. The Areg receives the address packet of a message at even cycles and the data packet at odd cycles. The opposite is true for Breg. Packets are stored in the format they are received from the WBIP with the RO bit appended to the address packet of each message.

Each slot connects to the following busses:

- The write bus (Wbus) is used to send data to the wait buffer from the FPC and connects to a wait buffer input port.
- The read bus (Rbus) is used by each slot for transmission of its message out of the wait buffer.
- The key bus (Kbus) contains the search key received from an MM port of the RPC.

[23] Since the destination of the message in WB_i is known on-chip at this cycle, this test can be refined to require only two empty slots in the queue that is the destination of the message.

[24] This structure requires each message to consist of a single address packet followed by a single data packet. Similar structures support messages containing a *fixed* even number of packets.

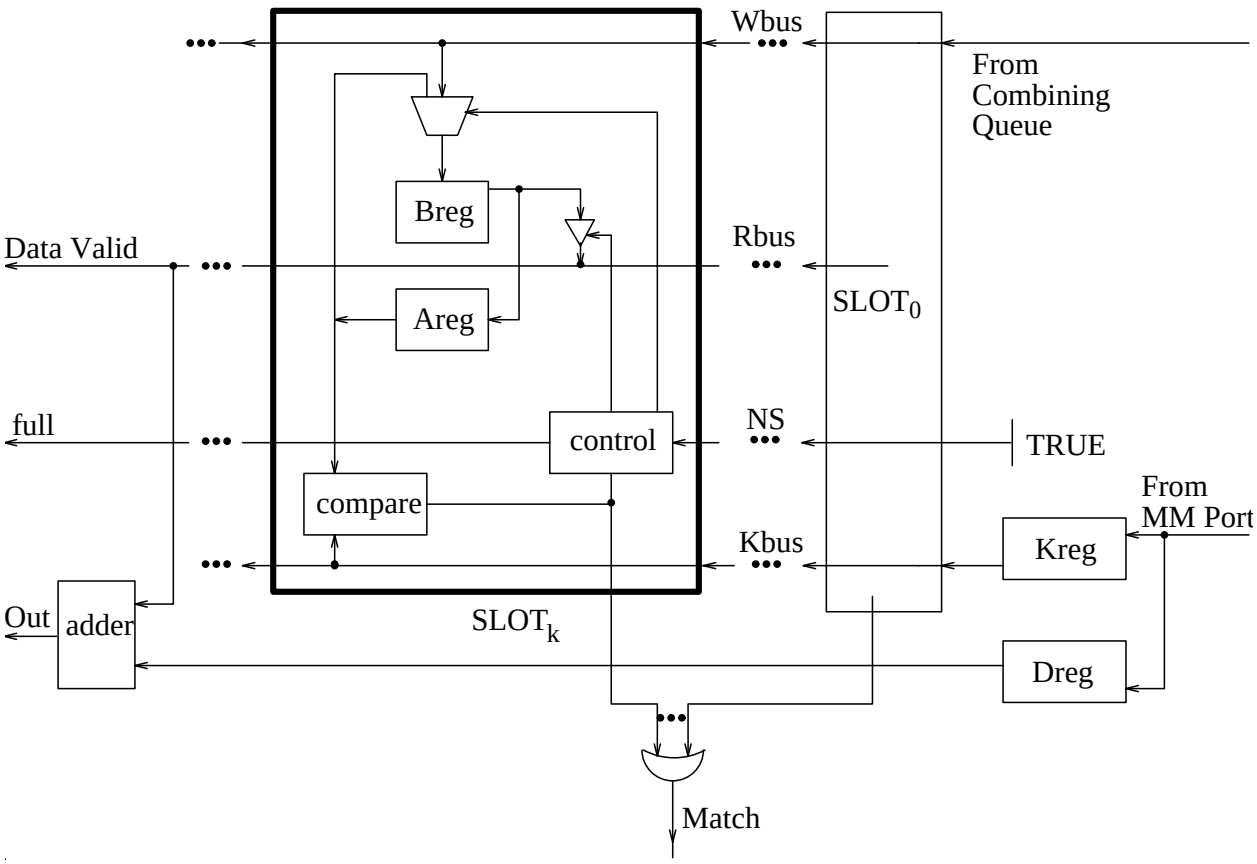

Figure 6 — Wait Buffer

The next-slot (NS) line is a one bit signal that is passed through all the slots in a daisy-chain fashion. It is used to select which slot will receive the next message from the FPC. Each slot computes

$$\mathrm{NS}_{out} := \mathrm{NS}_{in} \text{ and not } \textit{empty}$$

and the end of this signal, which has passed through all the slots, is the *full* line of the wait buffer.

An adder is used to generate the second response to an F&A operation by summing the data packet received from a slot and the data packet received from an MM. It passes address packets unchanged. Since the wait buffer is not required to respond until two cycles after it receives a matching memory response, the adder can take a full cycle.

The Dreg is connected to the MM port. It is loaded on odd cycles with the data packet of a message and presents that packet to the adder on the next cycle, which forwards the sum to the appropriate RPC queue on the cycle after that.

Each slot is *full* or *empty* depending on the DV bit in its Areg. During each pair of cycles $2t$ and $2t+1$, each wait buffer slot simultaneously performs the following operations:

(1) If DV is present on the Wbus, NS_{in} is on[25], and the slot is *empty*, the Areg is loaded with the address packet of the message at cycle $2t$. During cycle $2t+1$ the Breg receives the address packet and the Areg receives the data packet.

(2) If the slot is *full*, the data in the Areg and the data on the Kbus are compared during cycle $2t$. If the DV bit is present on the Kbus, the operation on the Kbus is an F&A, and the address and PE number are the same in the Areg as on the Kbus, the slot asserts *match*. (Note that only one slot can detect a match because the combination of address and PE number uniquely identify each combinable request in the network.) If a match is detected, the slot will present its message on the Rbus at cycles $2t+1$ and $2t+2$. It will also be marked as *empty* during cycle $2t+1$ so that it can begin accepting a subsequent message at cycle $2t+2$.

If any slot asserts *match* during cycle $2t$, the wait buffer will assert *match* at cycle $2t+1$. The packet presented by a slot to the Rbus on cycles $2t+1$ and $2t+2$ will be presented to Q_{1ij} on cycles $2t+2$ and $2t+3$, after being processed by the adder.

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

7.1.5.1. Pipelining Memory Requests The performance analyses given below and in Kruskal and Snir [83] assume that memory references are pipelined, i.e. that a PE may issue a request without waiting for previous ones to be acknowledged. However, since the queues in the network switches introduce stochastic delays, indiscriminate pipelining can lead to violations of the serialization principle (see Collier [81] and the references therein).

A simple way of preventing such behavior is not to pipeline memory requests directed at shared read-write variables. Accesses to private variables or read-only shared variables can be safely pipelined, as can instruction references (assuming that programs are not self-modifying). However, this policy is more conservative than necessary. A variable is classified as read-write shared based on its access pattern during the entire lifetime of the program. Since such a variable may, during a portion of the program's execution, be private or read-only shared, one can do better by dynamically determining whether to pipeline specific references. The status of a variable (private, read-only shared, or read-write shared) at each point in execution can be (conservatively) computed at compile time, and the compiler can generate the appropriate pipelined or nonpipelined request. When a shared variable changes status from read-write to read-only and all outstanding references are satisfied (this is likely to require synchronization), further references may be pipelined. See Shasha and Snir [86] for an analysis of the precise conditions needed to permit pipelining. Analogous considerations arise for determining cacheability, and our simulations (see McAuliffe [86]) show that permitting the cacheability status of a datum to vary during the program's lifetime increases performance considerably.

[25] As a performance optimization, a slot can load its registers whenever it is *empty*. Its DV bit is then set from the logical AND of the DV bit on the Wbus and the NS_{in} signal.

7.1.5.2. Limited Combining It is a concern that the complexity of the combining switches will make the network, already complicated by the message switching and buffering requirements, too cumbersome and too slow. Since combining memory requests affects only the performance of the machine—not the correctness of its computations—cost-performance considerations may dictate constructing a network with less than full combining. Some fall-back positions are:

- Do not combine requests at every stage of switches; this allows some stages to be faster.
- Use two separate networks: one fast and the other capable of combining. If requests to be combined are directed to a small number of locations, it is possible to build a combining network of linear cost (Harrison [84]).
- Combine only requests of the same type (e.g. loads with loads).
- Combine only pairs of messages at a switch. This allowed Snir and Solworth [84] to extend the systolic queue of Guibas and Liang [82] to obtain a queue that combines matching messages.

The NYU VLSI designs have only attempted to combine pairs of like requests. The IBM RP3 design includes a VLSI network that combines pairs of like requests and a faster bipolar network that does not support combining.

It is important to note that although the internal cycle time of the switches is important for today's technology, in the next 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.)

7.1.5.3. Hashing or Interleaving Memory Addresses A potential serial bottleneck is the memory module itself. If many PEs simultaneously requests a distinct word from the same MM, these N requests are serviced one at a time. Moreover, as with ''hot-spot'' references mentioned above, congestion backs up along the network and results in essentially all traffic (including that not directed at the busy MM) being severely impaired. 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. On average, the most requests directed at a single MM is (asymptotically) $\frac{log(N)}{log(log(N))}$ (see Gonnet [81] and, for a related analysis, Mehlhorn and Vishkin [83]). Furthermore, the probability that all MMs receive less than log(N) requests (asymptotically) exceeds 1–1/P(N) for any polynomial P. For many applications, sufficient hashing can be achieved by simply interleaving memory across the MMs. Interleaving is inadequate, for example, when arrays are declared with their dimensions a multiple of the number of MMs, in which case either entire rows or columns reside in a single MM. Having the compiler choose slightly larger dimensions would solve this particular problem.

One situation for which hashing cannot possibly be of any help is when concurrent references are directed at the same memory location. But this is precisely when combining is effective. We see, therefore, that these two techniques are complementary. Analogous situations have arisen in various software structures we have employed, see section 6.3.1 for an example.

7.2. 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 pipelining policy (i.e., whether to permit a request to enter the network before a previous request has returned), and cache management. The MNI (memory-network interface) is much simpler, performing only request assembly/disassembly and the addition operation necessary to support fetch-and-add. The MNI operations as well as the first two PNI functions are straightforward. The need to limit pipelining of memory requests was discussed above. Since the current switch design requires that a memory reference be determined by the PE number and memory address, the PNI must also prevent a PE from pipelining requests to the same memory location. Cache management is described below, when we discuss the entire memory hierarchy of the machine.

7.3. The Processors Naturally the processors need to support fetch-and-add. Moreover, to utilize fully 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 locked 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 would attempt to prefetch data sufficiently early to permit uninterrupted execution, and a cache designed for such processors would not lock up despite outstanding cache misses (see Kroft [81] for one such design).

If the latency remains an impediment to performance, we would hardware-multiprogram the PEs as in the CHOPP design (Sullivan *et al.* [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 problems of larger size, we view multiprogramming as a last resort.

The Ultracomputer design does not require homogeneous PEs. Therefore, a variety of special purpose processors, e.g. FFT chips, matrix multipliers, I/O processors (see section 7.5), voice generators, etc., can be attached to the network.

7.4. Memory Hierarchy

Although Ultracomputer memory appears as a single homogeneous store, familiar performance considerations dictate the implementation of a multi-level *memory hierarchy*, consisting of registers and caches at each PE, the central shared memory, and the secondary store or demand-paged virtual memory.

As mentioned previously the central memory of the machine is composed of multiple MMs, which are themselves fairly standard components. A significant design parameter of the overall machine is the wordsize of these MMs, since this determines the largest quantity that the hardware will reference atomically; larger units require software control for atomic access. The wordsize cannot be made arbitrarily large, however, since this impacts interleaving and, more significantly, Kruskal and Snir [83] have shown that network delays grow quadratically with message size. The RP3 generalizes the Ultracomputer design

and supports, in addition to interleaving (or hashing) memory across the MMs, the (runtime) specification of a portion of each MM that is to be addressed sequentially. The sequentially accessed portion of MM_i is well suited for memory that is to be referenced predominately by PE_i since a PE-MM pair can be packaged together so that messages from one to the other need not traverse the network.

It is necessary that the memory system be *coherent*, i.e. that the value returned from a load reflects the last value stored in the referenced address. Memory coherence problems can arise when parts of the hierarchy are replicated. We next describe the design of our cache and paging facility, both of which preserve memory coherence.

7.4.1. Cache The large memory latency introduced by the network can be largely avoided since most operations can be satisfied by a local memory. Moreover, local memory diminishes network traffic, thus reducing network latency due to conflicts.

The local memory can be implemented as a separately addressable memory or a cache. A separately addressable memory is cheaper and simpler to construct than a cache, but the machine becomes harder to program since the programmer or the support software must be aware of the two separate address spaces. Also, since the execution paths of a program may be unpredictable, a local memory may contain large amounts of extraneous data. A cache, on the other hand, is a ‘‘demand’’ device, i.e. only the data needed by the PE is brought into cache, and obsolete data is automatically evicted[26]. Many studies (see Smith [82]) have shown that a large cache can capture the overwhelming majority of references to cacheable variables in a uniprocessor environment. Our simulations have shown that this is also true in a tightly-coupled MIMD multiprocessor environment (see McAuliffe [86], which also contains an expanded presentation of the material in this section).

Indiscriminate use of a cache can introduce memory coherence violations. We do not know of any suitable hardware solution to the coherence problem for a large scale system that supports dynamic cacheability[27]. Presently proposed solutions suffer from various forms of serialization (Yen *et al.* [85]). We, therefore, propose the use of software to ensure run-time coherence[28].

Due to the negative impact of the interconnection network on the average memory access time, it is important for the cache to provide a low miss ratio and minimize the effect of a cache miss on subsequent PE memory requests. Many cache designs will not accept new requests until a miss is satisfied. We propose using instead a *lockup-free* cache (Kroft [81]), which may continue normal operation despite outstanding misses. The lockup-free attribute of the cache permits the PE to prefetch instructions and/or

[26]Depending upon the cache line size and program locality some extraneous data may reside in cache.

[27]That is, where a given data item may change its status from cacheable to non-cacheable and back during program execution. Our simulations have shown that dynamic cacheability offers a significant performance improvement.

[28]That is, special instructions or system service calls are defined, that when issued during program execution, explicitly alter the cacheability status of a data item.

data, with no delay incurred until they are needed, thus further masking network latency.

In a uniprocessor, increasing the cache line size (up to a point) has the effect of decreasing the cache miss ratio due to program locality, especially during process startup (Denning [72]). On the Ultracomputer, however, a large line size increases the amount of network traffic, and thus a less than optimal line size (with respect to miss ratio) is preferable. In addition, (temporarily) shared cacheable data does not have a high degree of spatial locality, whereas private data and instructions have a high degree of both spatial and temporal locality. Analysis and simulations, which take into account the network bandwidth and other design parameters of the Ultracomputer, suggest that with 32-bit PEs a line size of eight words is optimal.

Since the queuing delay in the network is quadratic in the size of a message, a cache line is stored in several consecutive MMs and is fetched by issuing several memory requests. The overhead of this scheme is the need for residency bits for partial cache lines.

Cache lines containing shared read-write date can reside in multiple caches provided that each modified datum is accessed by only one PE and only modified data are written to memory. The former condition must be guaranteed by the software that (dynamically) marks the data as cacheable. The second condition is easy to satisfy with a store-through cache; if a store-back policy is used, dirty bits must be maintained on the smallest addressable unit.

Since minimizing network traffic is a major objective of the cache design, a store-back policy intuitively appears to be a better policy than store-through[29]. However, a cache flush (necessary to update central memory when store-back is used) creates a burst of network traffic, which increases network delay for the remaining PEs; a steady, even if larger, amount of network traffic may be preferable. Measurements indicate that the largest number of variables modified by a process are shared. The largest number of store instructions executed, however, are directed to private data (e.g. those located on the stack). This behavior suggests that a combined update policy is preferable: store-back for private data and store-through for shared cacheable data. If the interconnection network and MMs provide sufficiently high bandwidth, a pure store-through policy gives only a small degradation when compared to the combined policy and is easier to implement.

7.4.2. Paging Although the Ultracomputer has a large central memory, one may still find it necessary to support demand paging to give the user a larger address space or to support a higher multiprogramming level. In this subsection, we describe several implementations; performance evaluations are underway and will be reported along with further details in Teller [86]. Here we assume that the status of a page (shared or private) is static; allowing a dynamic status slightly increases the complexity of these schemes.

[29]In fact, in Gottlieb *et al.* [83a] we claimed that store-back would be better. Simulations showed us to be wrong.

We group *k* contiguous MMs into a *cluster* (the value of k is a design parameter studied in Teller [86]) and associate with each cluster a *cluster controller* that either services page faults or assists faulting PEs in the servicing of a page fault[30].

We assume that each page frame is interleaved across a cluster of MMs. Thus, paging is local to each cluster. We further assume that each page is uniquely associated with a cluster of MMs. This can potentially cause uneven usage of clusters but a small page size and the placement of contiguous page frames at adjacent clusters will probably prevent this from occurring. This organization of memory is a compromise as it increases the incidence of memory conflicts (through coarser interleaving) while allowing us to design efficient mechanisms to manage virtual memory and input/output.

The PEs generate logical page addresses that must be translated into physical frame addresses. The location of the address translation mechanism strongly influences the implementation details of the paging system; there are two choices: It can be placed at each MM or at each PE.

The first approach requires that PE caches operate on virtual addresses and that each cluster controller has a directory with an entry for each virtual page that can reside in its cluster. To prevent the cluster controller from becoming a bottleneck, each MM has a directory with entries for the pages it currently contains (the MMs of a cluster have identical directories). When a page fault is detected by an MM, an appropriate message is sent to the cluster controller, inducing it to perform a page swap and to update the directory of each MM in its cluster[31]. PEs that initiate a page fault or that reference a page that is in the process of being brought into physical memory may be notified of the page fault or the memory response may just be delayed[32].

As usual, it must be quickly determined if a page is currently in memory, therefore, each MM is equipped with a directory cache, referred to as a *Translation Lookaside Buffer (TLB)*.

Performing address translation at the MMs is not without cost: Virtual addresses augmented with address space identifiers, which may be longer than physical addresses, must be transmitted across the network, increasing network traffic. Also, the optimal cluster size for paging may be bad for general I/O. We are, therefore, investigating the second option of locating the translation mechanism at the PEs.

When the address translation is performed at the PEs, clustering and fixed association of virtual pages with clusters are not necessary. However, because they facilitate page swapping I/O, we assume both.

---

[30]The cluster controller may be physically identical to an I/O processor that handles I/O for the cluster (see section 7.5) and is accessed by the PEs via the interconnection network.

[31]Since the directories of each MM in the cluster are identical, the cluster controller can broadcast revision information to the MMs via a bus.

[32]The schemes presented in this section can be modified so that the PEs service page faults in lieu of the cluster controller, thus increasing parallelism at the cost of increased network traffic.

In order to prevent two round trips through the network for each memory request (once to access the page table entry and again to access the word itself), a TLB is associated with each PE. When an entry is missing from its TLB, the PE accesses the page table associated with the cluster it is referencing; if the page is in physical memory, the PE updates its TLB, replacing the oldest entry by the new one, and accesses the word; otherwise, the cluster controller performs a page swap.

This approach can lead to memory incoherence since we are caching read-write shared objects, namely directory entries for shared pages. This problem is less severe with TLBs than it is with shared-data caches because modifications are less frequent. We have several solutions to this problem that employ timestamps, reference counts, and locking of pages in central memory.

One would expect that cooperating processes concurrently executing on several PEs would reference a common set of pages. Let p be the average number of active pages per PE, s the average number of PEs sharing a page, and c the cluster size. Then, the average number of active pages per MM is equal to pc/s. Thus, if the average amount of page sharing is larger than the cluster size, TLBs located at MMs will enjoy better hit ratios than those located at PEs. The fact that a PE TLB must be invalidated when a new process becomes resident also favors locating TLBs at the MMs. In addition, with an adequate level of page sharing, the amount of ‘‘bookkeeping’’ and the space required for TLBs will be less with the MM approach, and handling of dynamic status will be easier. These issues are under study.

7.5. Input/Output

Physical I/O on a large scale parallel processor presents many problems. There are often large amounts of data that must be moved quickly, which can significantly interfere with the execution of the PEs. In the Ultracomputer, I/O can be supported in two ways:

- ‘‘Front-end’’ I/O performed by the PEs. Some of the PEs may be dedicated I/O processors that are connected to I/O devices.
- ‘‘Back-end’’ I/O performed by other processors that use a second port on the MMs.

The first approach has the advantage that it can be implemented without any additional logic: the same mechanism that allows a PE to access memory provides memory access to the I/O processor. Note, however, that the interconnection network has been designed to support memory accesses to small blocks (words or cache lines) of data scattered through memory, whereas I/O requests are typically for large contiguous blocks and thus may not be well served by the same network used by the PEs. One is led, therefore, to consider the latter approach: incorporating a second, ‘‘back-end’’, network that connects the (dual-ported) MMs to the I/O devices.

If standard I/O devices are to be used, the back-end I/O network for a clustered memory consists of two levels. A cluster switch (which might be a bus) cyclically connects successive MMs within the cluster to the cluster I/O port(s). A second circuit-switched network allows each cluster I/O port to connect to each I/O device.

A more aggressive approach would be to use disks that concurrently access all the tracks on a given cylinder (i.e. all the surfaces of the disk). Then one could set the MM cluster size equal to the number of surfaces and interleave the disk across the surfaces so that the kth MM in a cluster always accesses the kth surface on a disk. In this way all the MMs in a cluster could read or write a disk at once.

We realize that the mechanisms suggested here will not provide I/O capacity that matches the aggregate processor speed; a 4096-PE machine will not have 4096 disks. However, many interesting computational problems require a processing time that increases faster than linearly in the size of the problem. Problems that seem I/O-bound are often small sized instances, where the nonlinear processing time may be dominated by the linear, but larger, I/O time. If the instance size is increased, the I/O bottleneck may disappear.

7.6. Reflection and Refraction

Although interprocess communication can easily be supported by the shared memory, we propose a new *reflection* operation to increase efficiency in some cases. We add a new memory operation, with format *send(addr)*, where *addr* is the shared memory address of the *reflector* of a process. The reflector contains the current PE number of the process; when the process is not running, its reflector is disabled. When a send request directed at an enabled reflector arrives at the MM, it is forwarded to the PE whose number is stored therein and a positive acknowledgment is returned to the sender. If the reflector is disabled, a negative acknowledgment is returned. When the forwarded request arrives at the referenced PE, an interrupt is generated.

Note that since there is a unique path between any MM and any PE, the acknowledgment of a disable operation issued by PE_i must travel the same route that a reflected message to PE_i would take. Therefore, when the acknowledgment is received, there are no more outstanding reflected messages and the process may migrate. Send operations cannot be combined by the network.

The same mechanism can be used for communication between the computational PEs and the specialized I/O processors employed in front-end I/O. If back-end I/O is implemented, reflection is replaced by *refraction*. A further extension of the reflection/refraction mechanism allows broadcasting a message to all PEs by duplicating the message at each switch in the MM-to-PE direction. A future paper will discuss broadcasting messages to a subset of the PEs.

7.7. Machine Packaging

We estimate that a machine built in 1990 would require four chips for each PE-PNI pair, nine chips for each MM-MNI pair and two chips for each 4-input-4-output switch (which replaces four of the 2×2 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 2×2 switches into $sqrt(N)$ ‘‘input modules’’ and $sqrt(N)$ ‘‘output modules’’. An input module consists of $sqrt(N)$ network inputs and the $sqrt(N)(lg\ N)/4$ switches that can be accessed from these inputs in the first $(lg\ N)/2$ stages of the network[33]. An output module consists of $sqrt(N)$ network outputs and the $sqrt(N)(lg\ N)/4$ switches that can be accessed from these outputs in the last half of the network. Moreover, as noted by Wise [81], it is possible to arrange the switches of each module so that, between any two successive stages, all lines have the same length (Figure 7). 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 8. The same strategy can be used for networks built of k×k switches. (Figures 7 and 8 are reprinted with permission from Wise [81].)

We propose using this layout for a 4K processor machine constructed from the chips described at the beginning of this section. This 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 4×4 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. Bianchini and Bianchini [82] have considered machines built from single chip 2×2 switches. Their analysis shows that an air cooled 4096 PE ultracomputer can easily be packaged into a 5’x5’x10’ enclosure.

[33]We use *lg* for the base 2 logarithm.

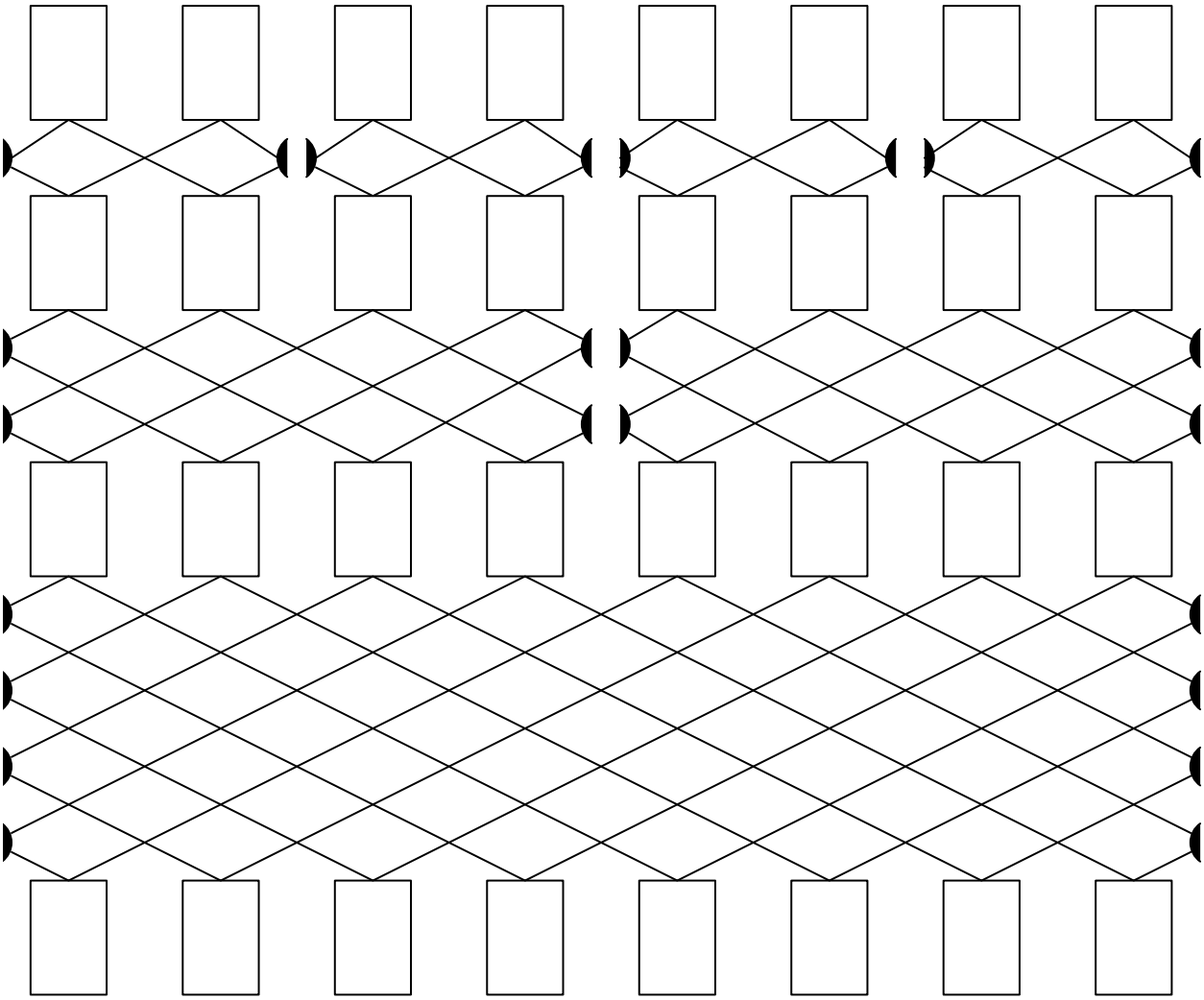

Figure 7. Layout of network on boards.

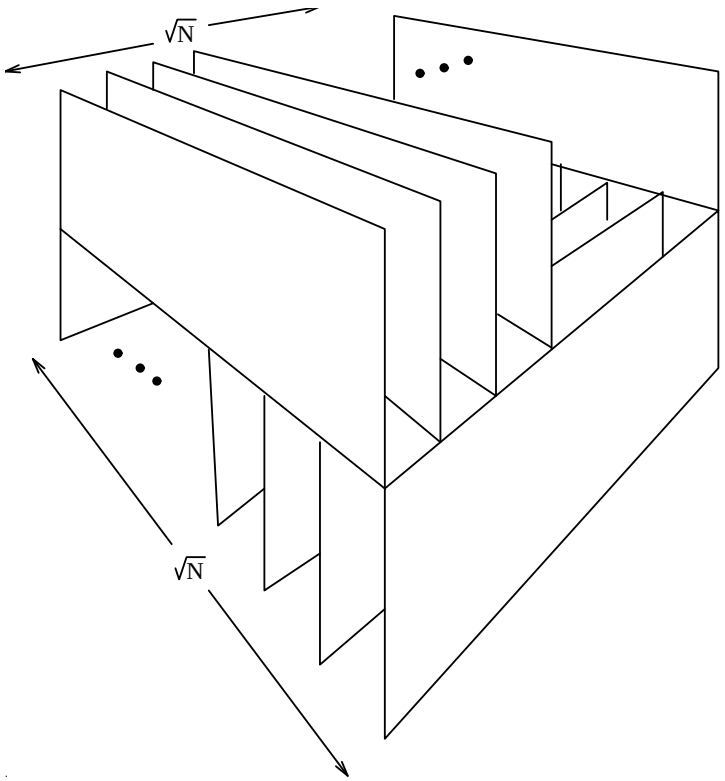

Figure 8. Packaging of network boards.

8. 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.

8.1. Performance Analysis

Although each switch in the network requires a significant amount of hardware, it appears feasible to implement a 2×2 switch on one chip using today's technology. Further, we believe it will be feasible in 1990 technology to implement 4×4 or even 8×8 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 therefore be 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 ns.). If each message transmitted through the network consists of approximately 100 bits (64 bits of data), then a 2×2 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 2×2 switches by k×k 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:

k the size of the switch. Recall that a k×k switch requires 4k lines.

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.)

d the number of copies of the network that are used.

The chip bandwidth limits the k/m ratio, which we may thus take to be independent of the network configuration. Note that for any k a network with n inputs and n outputs can be built from $\frac{n \ lg \ n}{k \ lg \ k}$ k×k 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. Thus the cost of a configuration is $\frac{d(n \ lg \ n)}{k \ lg \ k}$ (we are neglecting the small cost of interfacing the d copies of the network).

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

- Requests are not combined.
- Requests have the same length.
- Queues are of infinite size.
- Requests are generated at each PE by independent, identically distributed, time-invariant random processes.
- 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

$$1 + \frac{m^2 p(1-1/k)}{2(1-mp)}$$

cycles (see Kruskal and Snir [83]; similar results can be found in Jacobsen [77] and Dias and Jump [81]). The average network traversal time (in one direction) T equals the number of stages times the switch delay plus the setting time for the pipe, i.e.

$$T = \frac{lgn}{lgk}\left[1 + \frac{m^2 p(1-1/k)}{2(1-mp)}\right] + m - 1.$$

Let us note the following facts:

(1) The network has a capacity of 1/m messages per cycle per PE. That is each PE cannot enter messages at a rate higher than one per m cycles, and conversely the network 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 queuing 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 queuing delay for a switch with a multiplexing factor of m is roughly the same as the queuing 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, when using k×k switches, the time multiplexing factor m equals k. 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 k×k switches is

$$T = lg\ \frac{n}{lg\ k}\left[1 + \frac{k(k-1)p}{2(d-kp)}\right] + k - 1$$

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 9 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, a duplicated network composed of 4×4 switches yields the best performance (but see the next paragraph for a preliminary evaluation for networks consisting of two-chip switches). A network with 8×8 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. Of course a final determination of an optimal configuration 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 8×8 impractical for a 4K PE parallel machine.

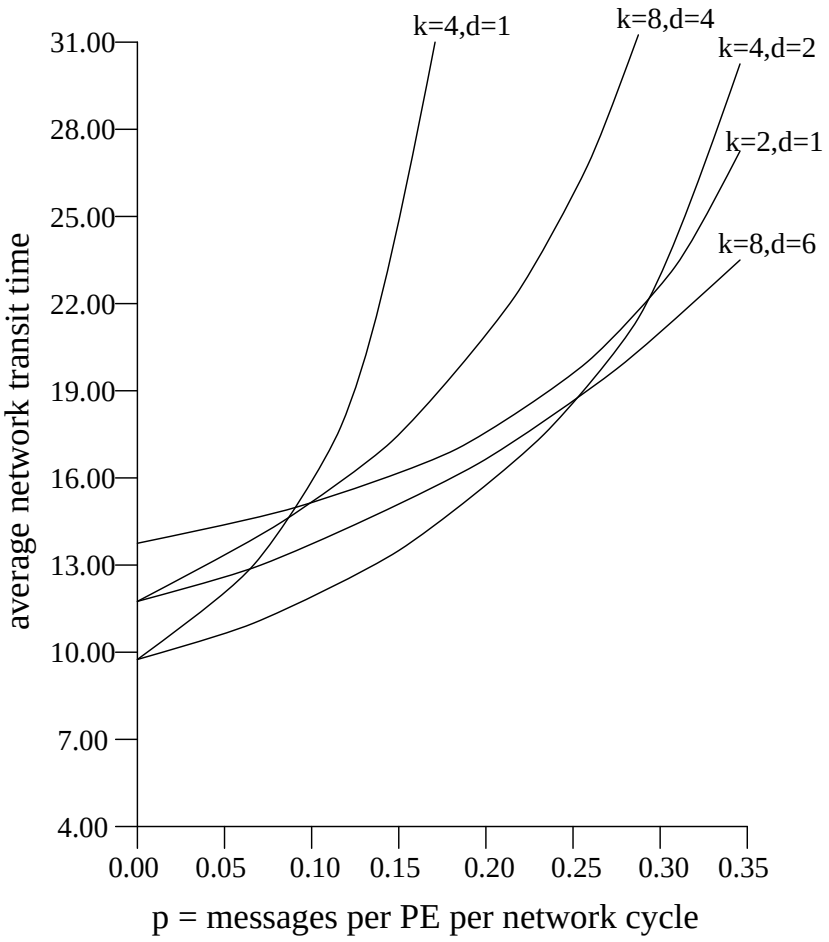

Figure 9. Transit times for different configurations.

The previous discussion assumed a one chip implementation of each switch. By using the two chip implementation described in section 7, 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 4×4 switches, each switch consisting of two chips.

We now return to the five assumptions listed above. The 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 combined messages do not each traverse the entire network.

Simulations have shown that queues of modest size (≤8) give 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[34].

8.2. 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 have run parallel scientific programs under simulation both to measure the speedup obtained by parallelism and to judge the difficulty involved in creating parallel programs. As mentioned in section 2 and described in Kalos [81], the incremental effort required to obtain a parallel program was always significantly less that what was needed to obtain an equivalent serial program. For many of the experiments, accessing memory on the simulated machine was assumed to require just a single cycle as in our computational model described in section 2; other runs used an approximation to the proposed network design rather than an assuming single cycle memory access (see Gottlieb [80] and Snir [81]). Since an accurate network simulation would be very expensive, we implemented instead a multi-stage queuing system model with stochastic service time at each stage, parameterized to correspond to a network with six stages of 4×4 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 network 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 monitored the amount of network traffic generated by several scientific programs under the pessimistic assumption 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.

[34]Of course memory access to ‘‘hot spots’’ is not diminished by hashing, but the impact is reduced significantly by combining.

(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. In these simulations the number of requests to central memory (CM) were comfortably below what the network can support and indeed the average access time was close to the minimum possible. (Since each PE was a CDC 6600-type CPU, a load-store architecture, most instructions involved register-to-register operations.) 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[35]. 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 central memory was significantly lower than the central memory access time.

program number	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

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 execution.

9. Scientific Programming and Estimated Performance

As indicated in section 8.2 we used an instruction level simulator to study parallel variants of scientific programs. Applications already studied include radiation cascade, incompressible fluid flow within an elastic boundary, atmospheric modeling, polymer simulations, transport Monte Carlo, and Monte Carlo simulation of fluid structure.

[35]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.

The goals of our simulation studies were, 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

$$T(P,N) = aN + dN^3/P + W(P,N)$$

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 $\max(N,P^{.5})$. 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 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

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

Reduction of Matrices to Tridiagonal Form					
	Number of Processors				
N	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.

Reduction of Matrices to Tridiagonal Form					
N	Number of Processors (without waiting time)				
	16	64	256	1024	4096
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.

10. Conclusion: Our Status and Future Plans

10.1. Prototype Hardware

We have constructed several bus-based prototypes and currently operate four reliable (Ultra II) systems, the two largest each containing eight PEs and 8MB of central memory. The processors are 10 MHz Motorola MC68010 augmented with Weitek 1164/65 floating point units, 32KB 2-way set associative caches, and hardware support for fetch-and-add. Each processor supports two serial ports; one of these sixteen ports is used as the system console. The central memory is packaged as four 2MB MMs each containing an adder used to support fetch-and-add. A second port on the shared memory is connected to an I/O laden PDP-11 that, in addition to serving as an all-purpose I/O controller, connects the prototype to the NYU network, and hence to the arpanet as well. These prototypes are in routine use for software development, algorithms and application studies, and scientific production, both within and outside NYU. We have delivered a prototype to the NYU robotics laboratory, where it will be used for real time robot control. The university has offered a course on programming MIMD shared memory computers for which students were required to develop, run, and analyze programs on the prototypes. This course has been valuable in disseminating the methodologies of programming the ultracomputer in C, FORTRAN, and LISP in a wide variety of applications. The reliability of our hardware and software was confirmed.

We plan to build a more powerful system based on Motorola 68030/68882 or other high performance processors and floating point coprocessors. Our current target is a 64 processor configuration, including a VLSI combining network for accessing a 256 megabyte shared memory. It will provide for higher performance and, by virtue of the larger number of processors, for a more exigent test of the qualities of different approaches to parallel processing.

10.2. Systems Software

As described in section six, we have implemented a symmetric highly parallel extension of UNIX

(Version 7), dubbed ‘‘Symunix’’[36] that is in routine production use on our hardware prototypes. To obtain a functioning operating system as soon as possible and thus allow early experimentation with parallel application programs, we first implemented a serialized ‘‘master-slave’’ version of UNIX, which we retired when Symunix became available. The master-slave system should be thought of as a serial program (running on multiple processors) that supports concurrent user programs; whereas Symunix is itself a highly parallel program using fetch-and-add based algorithms largely free of critical sections. The spawn system call discussed in previous sections has been implemented and is usable from C and FORTRAN.

We have designed Symunix II, a highly parallel upward compatible extension of Berkeley UNIX (4.3 BSD) for both our own hardware and the IBM RP3. Unlike the original Symunix, the new version is a significant functional extension of its base: the process and memory model supported go well beyond UNIX. Symunix II will be described in future reports. The implementation effort is well underway; signs of life have been observed on Ultra II and a simulator for RP3.

Recently, we have begun to address the important issue of the programming environment seen by users of our prototypes and future users of the RP3. We plan to expand this effort considerably. To date we have modified the C and FORTRAN compilers to support parallel user programs in a style similar to that given in section three. In particular, the extended languages permit shared variables, parallel loops, and parallel blocks. See Grob and Lipkis [86] for a further discussion of issues in parallel programming. In addition, various program libraries are available, which contain coordination routines such as barrier synchronization and counting semaphores, routines for concurrent access to important data structures such as queues, software support for the Weitek floating point hardware, and other items as well. A user-mode tasking library, which will provide lightweight processes, is under development. Both a compiler and an interpreter for a parallel variant of lisp, entitled ZLISP (Dimitrovsky [86]) are now in use as is pdb (Berke [86]), a debugger for parallel user programs extending the standard unix assembly language debugger adb. Ultraprolog is under development: A compiler translating a variant of Prolog supporting ‘‘and parallelism’’ into Zlisp has just been written. The issue of how best to support ‘‘or parallelism’’ has not yet been settled.

10.3. Application Studies and Machine Simulation

We have continued to create parallel programs for solving important problems both by parallelizing existing serial solutions and by writing parallel programs from scratch. Programs previously run under simulation are being adapted to run on the prototypes. New applications in Monte Carlo and Fluid Dynamics are being studied and a natural language parser has been converted from (serial) Franz lisp to ZLISP. External users, notably at Sandia National Laboratory, are using the machines.

[36]The most natural name for an ultracomputer version of unix is probably ‘‘Ultrix’’, a name we thought of years ago but never pursued, the term OS being adequate for our internal use. It served us right when DEC trademarked Ultrix.

A research program on the generation of pseudo-random numbers with the stringent properties required for highly parallel MIMD computers has begun (Percus and Kalos [87]). The prototypes are being used extensively for testing of the experimental sequences of sequences.

We are investigating alternative implementations of combining and their effect on network bandwidth and latency in the presence of memory hot spots.

10.4. Architecture Research

Research in comparative assessment of advanced architectures will also continue. Of particular interest will be massive I/O for parallel machines; the possibility of heterogeneous Ultracomputers with specialized processors at some nodes; architectures with more complex memory hierarchies, e.g., with caches distributed in the network; the requirements of very large processor ensembles (greater than roughly 2^{16} PEs); and generalizations of the distributed computing now performed by the combining networks. The last will include consideration of architectural support for higher level primitives (than Fetch-and-add) that may prove more efficient in supporting communication and synchronization protocols. Several examples appear in Harrison [86], including the Test-Decrement-Retest (TDR) macro used in our highly parallel queue and readers-writers algorithms. We will also consider supporting still higher level primitives that may prove especially efficient for high level languages and powerful programming environments.

10.5. VLSI Design

In preparation for the design of a complete combining switch chip, we have designed several simpler chips, which have been fabricated by DARPA's MOSIS facility and tested at NYU.

We have received functional 11-bit wide 2x2 non-combining switch chips containing approximately 7500 transistors and fabricated in 3-micron NMOS. These parts operate at a clock speed of 23mHz with a propagation delays from clock to output of approximately 25ns. Power dissipation is approximately 1.5W. A 4x4 test network was constructed using four of these parts and functioned correctly.

We have also had a 6-bit wide portion of the FPC (without the adder) for a 2x2 combining switch fabricated in 4-micron NMOS. These switches are composed of four 1-input combining queues and have performance and power dissipation similar to the non-combining switches.

Since the final combining switches must be at least 32-bits wide and air-cooled, we have redirected our design effort to the newly available scalable double-metal CMOS process, which promises minimum feature sizes as small as 1.6 microns. We have received from MOSIS and tested a functioning preliminary version of a 35-bit wide non-combining switch that uses this technology. A detailed VLSI design for the next version of this chip, which also supports loads and stores of doublewords, is complete. It meets both the MOSIS and IBM design rules. We have just received chips from MOSIS that function correctly providing the clock overlaps are within certain ranges. We believe IBM will also fabricate the chip (but using state of the art technology), and hope that the resulting parts will be used in the IBM RP3

hardware. It was for this application that doubleword support was added.

The key subcomponent of the memory to processor direction is an associative memory needed for the decombining of memory responses. We will soon test CMOS chips containing this subcomponent.

Our plans include construction of a network from the recently received CMOS non-combining chips. Note that, when combining is not supported, there is no communication between the FPC and RPC and that these two components are identical. Thus, we could construct a bidirectional non-combining network from the one chip type already submitted for fabrication. In addition, we plan to develop the full FPC and RPC and described above and then construct a combining network. We will standardize on the pinouts early in this process so that the combining and noncombining chips will be compatible. Upgrading a network to support combining will consist of simply replacing switch chips.

References

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

John R. Allen and Ken Kennedy, ‘‘PSC: A Program to Convert FORTRAN to Parallel Form’’, in *Supercomputers: Design and Application*, Kai Hwang, Ed., IEEE Computer Science Press, 1984.

Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante, ‘‘An Overview of the PTRAN Analysis System’’ *J. Parallel and Distributed Comp.*, 1988, pp. 617-640.

Maurice J. Bach and Steven J. Buroff, ‘‘Multiprocessor UNIX Operation Systems’’, AT&T Bell Laboratories Technical Journal **63**, no. 8, pp. 1733-1749, October 1984.

R. Baron, Richard Rashid, E. Siegel, A. Trevanian, and M. Young, ‘‘Melange: A Multiprocessor-Oriented Operating System Environment’’, Technical Report, Department of Computer Science, Carnegie-Mellon University, 1985.

Wayne Berke, ‘‘Pdb–An Ultracomputer Debugger for Parallel Programs–User’s Guide’’, Ultracomputer Documentation Note #3, Courant Institute, NYU, 1986.

Ronald Bianchini and Ronald Bianchini Jr., ‘‘Wireability of the NYU Ultracomputer’’, Ultracomputer Note #43, Courant Institute, NYU, 1982.

Alan Borodin and John E. Hopcroft, ‘‘Routing, Merging, and Sorting on Parallel Models of Computation’’, *14th Annual ACM Symposium on Theory of Computing*, pp. 338-344, 1982.

Per Brinch-Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

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

D. Chazan and Willard Miranker, ‘‘Chaotic Relaxation’’, *Lin. Alg. and Appl.* **2**, pp. 199-222, 1969.

William W. Collier, ‘‘Principles of Architecture for Systems of Parallel Processes’’, IBM Technical Report TR00.3100, March 1981.

M. Conway, ‘‘A Multiprocessing System Design’’, *AFIPS 1963 FJCC*, (Spartan Books, New York).

J. Davies, ‘‘Parallel Loop Constructs for Multiprocessors’’, Technical Report UIUCDCS-R-81-1070, University of Illinois, Urbana, Illinois, May 1981.

Denelcor, *Heterogeneous Element Processor, Principles of Operation*, 1981.

Peter J. Denning, ‘‘On Modeling Program Behavior’’, *Proc. SJCC.* pp. 937-944, 1972.

Jack B. Dennis and E.C. Van Horn, ‘‘Programming semantics for multiprogrammed computations’’, *Communications of the ACM* **9**, no. 3, March 1966.

Narsingh Deo, C.Y. Pang, and R.E. Lord, ‘‘Two Parallel Algorithms for Shortest Path Problems’’, in *Intl. Conf. on Parallel Proc.* , pp. 244-253, 1980.

Daniel Dias and J. Robert Jump, ‘‘Analysis and Simulation of Buffered Delta Networks’’, *IEEE Trans. Comput.* **C-30** pp. 273-282, 1981.

Susan Dickey, Allan Gottlieb, and Richard Kenner, ‘‘Using VLSI to Reduce Serialization and Memory Traffic in Shared Memory Parallel Computers’’, in *Advanced Research in VLSI: Proc. of the Fourth MIT Conf.*, Charles E. Leiserson, (ed.), 1986a.

Susan Dickey, Richard Kenner and Marc Snir, ‘‘An Implementation of a Combining Network for the NYU Ultracomputer’’, Ultracomputer Note 93, Courant Institute, NYU, 1986b.

Susan Dickey, Richard Kenner, Marc Snir, and Jon Solworth, ‘‘A VLSI Combining Network for the NYU Ultracomputer’’, *Proc. Intl. Conf. on Comp. Design*, 1985.

Edsger W. Dijkstra, ‘‘Co-operating Sequential Processes’’, in *Programming Languages*, F. Genuys (ed.), Academic Press, New York, pp. 43-112, 1968.

Isaac Dimitrovsky, ‘‘ZLISP Reference Manual’’, Ultracomputer Documentation Note #4, Courant Institute, NYU, 1986.

E. Draughon, Ralph Grishman, Jacob T. Schwartz, and A. Stein, ‘‘Programming Considerations for Parallel Computers’’, *IMM 362*, Courant Institute, NYU, November 1967.

Jan Edler, Allan Gottlieb, Clyde P. Kruskal, Kevin McAuliffe, Lawrence Rudolph, Marc Snir, Patricia Teller, and Jim Wilson, ‘‘Issues Related to MIMD Shared Memory Computers: The NYU Ultracomputer Approach’’, *Proc. 12 Annual Comp. Arch. Conf.*, 1985a.

Jan Edler, Allan Gottlieb, and Jim Lipkis, ‘‘Considerations for Massively Parallel UNIX Systems on the NYU Ultracomputer and IBM RP3’’, *Proc. USENIX Assoc. Winter Conf.*, 1986.

Jan Edler, Allan Gottlieb, and Jim Lipkis, ‘‘Operating System Considerations for Large-Scale MIMD Machines’’, *Computers in Mech. Eng.*, 1985b.

K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, ‘‘The notion of Consistency and Predicate Locks in a Database System’’, *CACM*, pp. 624-633, November 1976.

William A. Felton, Gerald L. Miller, and J. Michael Milner, ‘‘A UNIX System Implementation for System/370’’, AT&T Laboratories Technical Journal **63**, no. 8, pp. 1751-1767, October 1984.

M.J. Flynn, ‘‘Very High-Speed Computing systems’’, *IEEE Trans.* **C-54**, pp. 1901-1909, 1966.

Steven Fortune and James Wyllie, ‘‘Parallelism in Random Access Machines’’, *10th ACM Symp. on Theory of Comp.*, pp. 114-118, 1978.

Geoffrey C. Fox and S.W. Otto, ‘‘The Caltech Concurrent Computation Program–A Status Report’’, *Computers in Mechanical Engineering*, ASME and Springer-Verlag, March 1986.

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

Gaston H. Gonnet ‘‘Expected Length of the Longest Probe Sequence in Hash Code Searching’’, *JACM*, pp. 289-304, April 1981.

J.A. Gosden, ‘‘Explicit Parallel Processing Description and Control in Programs for Multi and Uniprocessor Computers’’, *Proc. AFIPS 1966 Fall Joint Comp. Conf.*, Spartan Books, New York, pp. 651-660, 1966.

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*, pp. 16-24, October 1981.

Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Lawrence Rudolph, and Marc Snir, ‘‘The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer’’, *IEEE Trans. Comp.*, pp. 175-189, February 1983a.

Allan Gottlieb, Boris D. Lubachevsky, and Lawrence Rudolph, ‘‘Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors’’, *ACM TOPLAS* **5**, pp. 164-189, April 1983b.

Allan Gottlieb and Jacob T. Schwartz, ‘‘Networks and Algorithms for Very Large Scale Parallel Computations’’, *Computer* 15, pp. 27-36, January 1982.

Lori Grob and Jim Lipkis, ‘‘Approaches to Parallel Programming on Multiprocessors’’, *Proc. EUUG Autumn Workshop on Distributed UNIX Systems*, September 1986.

Leo J. Guibas and Frank M. Liang, ‘‘Systolic stacks, queues, and counters’’, *Conf. on Advanced Research in VLSI*, MIT, January 1982.

Malcolm C. Harrison, ‘‘Synchronous Combining of Fetch-and-add Operations’’, Ultracomputer Note #71, Courant Institute, NYU, 1984.

Malcolm C. Harrison, ‘‘The Add-and Lambda Operation: An Extension of F&A’’, Ultracomputer Note #104, Courant Institute, NYU, 1984.

E.A. Hauck and B.A. Dent, ‘‘Burroughs’ B6500/B7500 Stack Mechanism’’, *AFIPS 1968 SJCC*, pp. 245-251. Also in Daniel P. Siewiorek, C. Gordon Bell, and Alan Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, pp. 244-250, 1982.

Robert G. Jacobsen and David P. Misunas, ‘‘Analysis of structures for packet communication’’, *Intl. Conf. on Parallel Proc.*, 1977.

Steven D. Johnson, ‘‘Connection Networks for Output-Driven List Multiprocessing’’, Technical Report 114, Computer Science Dept., Indiana University, 1981.

Harry Jordan, ‘‘Special Purpose Architecture for Finite Element Analysis’’, *Proc. IEEE Intl. Conf. on Parallel Proc.*, 1978.

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

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

Ken Kennedy, ‘‘Automatic Transformation of FORTRAN Programs to Vector Form’’, Technical Report 476-029-4, Department of Mathematical Sciences, Rice University, October 1980.

David Klappholz, private communication, 1981.

David Klappholz, ‘‘Parallelized Process Scheduling for a Tightly-Coupled, MIMD Machine’’, Technical Report, Division of Computer Science, Polytechnic Institute of New York, 1982.

Donald E. Knuth, *The Art of Computer Programming*, **v. 1,** Fundamental Algorithms, Addison-Wesley, 1968.

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

David Korn and Norman Rushfield, ‘‘Washcloth Simulation of Three-Dimensional Weather Forcasting Codes’’, Ultracomputer Note #55, Courant Institute, NYU, 1983.

David Kroft, ‘‘Lockup-Free Instruction Fetch/Prefetch Cache Organization’’, *The 8th Annual Symposium on Computer Architecture*, pp. 81-88, 1981.

Clyde P. Kruskal, ‘‘Upper and Lower Bounds on the Performance of Parallel Algorithms’’, Dissertation, Courant Institute, NYU, 1981.

Clyde P. Kruskal and Marc Snir, ‘‘The Performance of Multistage Interconnection Networks for Multiprocessors’’, *IEEE Trans. Computers* **C-32**, pp. 1091-1098, 1983.

Clyde P. Kruskal and A. Weiss, ‘‘Allocating Independent Subtasks on Parallel Processors’’, *ICPP*, pp. 236-240, 1984.

Clyde P. Kruskal, Larry Rudolph, and Marc Snir, ‘‘Efficient Synchronization on Multiprocessors with Shared Memory’’, *ACM TOPLAS* **10** No. 4, pp. 579-601, Oct., 1988.

David Kuck and David A. Padua, ‘‘High Speed Multiprocessors and Their Compilers’’, *Proc. Intl. Conf. Parallel Proc.*, 1979.

H.T. Kung, ‘‘The Structure of Parallel Algorithms’’, in *Advances in Computers* **19** M.C. Yovits (ed.), Academic Press, pp. 65-112, New York, 1980.

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

Gyungho Lee, Clyde P. Kruskal, and David J. Kuck, ‘‘The Effectiveness of Combining in Shared Memory Parallel Computers in the Presence of ‘Hot Spots’’’ *Proc. ICPP*, 1986, pp. 35-41

Stephen F. Lundstrom and George H. Barnes, ‘‘A Controllable MIMD Architecture’’, *Proc. Intl. Conf. Parallel Proc.*, pp.19-27, 1980.

Kevin McAuliffe, *Analysis of Cache Memories in Highly Parallel Systems*, Ph.D. thesis, Courant Institute, NYU, 1986.

Kurt Mehlhorn and Uzi Vishkin, ‘‘Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories’’, *Acta Informatica* **21** pp. 339-374, 1984.

Gregory F. Pfister and V. Alan Norton, ‘‘‘Hot Spot’ Contention and Combining in Multistage Interconnection Networks’’, *IEEE Transactions on Computers*, pp. 943-948, October 1985.

Ora E. Percus and Malvin H. Kalos, ‘‘Random Number Generation for Ultracomputers’’, Ultracomputer Note #114, Courant Institute, NYU, 1987.

George Radin, ‘‘The 801 Minicomputer’’, *Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47, 1982.

David P. Reed and Rajendra K. Kanodia, ‘‘Synchronization with Eventcounts and Sequencers’’, pp. 115-123, *CACM* **22**, 1979.

Gary Rodrigue, E. Dick Giroux, and Michael Pratt, ‘‘Perspectives on Large-Scale Scientific Computing’’, *IEEE Computer* **13**, pp. 65-80, October 1980.

Lawrence Rudolph, ‘‘Software Structures for Ultraparallel Computing’’, Ph.D. thesis, Courant Institute, NYU, February 1982.

Jacob T. Schwartz, ‘‘Preliminary Thoughts on Ultracomputer Programming Style’’, Ultracomputer Note #3, Courant Institute, NYU, 1979.

Jacob T. Schwartz, ‘‘Ultracomputers’’, *ACM TOPLAS*, **2**, 4, pp. 484-521, October 1980.

Dennis Shasha and Marc Snir, ‘‘Efficient and Correct Execution of Parallel Programs That Share Memory’’, *ACM TOPLAS*, **10**, 2, pp. 282-312, April 1988.

Alan Jay Smith, ‘‘Cache Memories’’, *ACM Computing Surveys*, **14**, pp. 473-530, September 1982.

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

Marc Snir, ‘‘On Parallel Search’’, *Proc. Principles of Distributed Computing Conf.*, pp. 242-253, August 1982.

Marc Snir and Jon Solworth, ‘‘The Ultraswitch – A VLSI Network Node for Parallel Processing’’, Ultracomputer Note #39, Courant Institute, NYU, 1984.

Harold S. Stone, ‘‘Parallel Computers’’, in *Introduction to Computer Architecture*, Harold S. Stone, ed., Chicago, IL, SRA, pp. 318-347, 1980.

Herbert Sullivan, Theodore Bashkow, and David Klappholz, ‘‘A Large Scale Homogeneous, Fully Distributed Parallel Machine’’, *Proc. of the 4th Annual Symp. on Comp. Arch.*, pp. 105-124, 1977.

Patricia J. Teller, ‘‘The Feasibility of Demand Paging in an MIMD Shared Memory Parallel Processor’’, Ph.D. thesis, Courant Institute, NYU, 1986, in preparation.

James Wilson, ‘‘Task and Memory Management for MIMD Shared Memory Parallel Computers’’, Ph.D. thesis, Courant Institute, NYU, 1986, in preparation.

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, pp. 186-195, 1981.

Wei C. Yen, David W.L. Yen, and King-Sun Fu, ‘‘Data Coherence Problem in a Multicache System’’, *IEEE Trans. Computers* **C-34,** pp. 56-65, January 1985.

Table of Contents

1. Introduction 1
2. Machine Model 2
2.1. The Idealized Paracomputer or WRAM 3
2.2. The Fetch-and-add Operation 4
2.3. The General Fetch-and-ϕ Operation 5
2.4. The Yet More General Read-Modify-Write Operation 6
2.5. The Power of Fetch-and-add 6
2.6. Alternate machine models 8
3. Parallel Programming 9
3.1. Levels of Parallel Control 9
3.2. Parallel Constructs 10
3.2.1. Control Structures 11
3.2.2. Shared Variables 11
3.2.3. Barrier Synchronization 12
3.3. Implementation of Parallel Constructs 12
3.4. Performance Issues in Parallel Control 13
3.5. Performance Issues in Barrier Synchronization 14
4. Process scheduling 14
4.1. The ‘‘Self-Service’’ Paradigm 14
4.2. Scheduling for a Spectrum of Program Styles 16
5. Operating System Interface 17
5.1. Process Management 18
5.1.1. System Calls 18
5.1.2. Low-Overhead Parallel Threads 19
5.2. User Memory Structure 20
5.2.1. Object Files 20
5.2.2. Shared Data Segment 20
5.2.3. Private Data Segment 21
5.2.4. Program Stack Segment 21
5.2.5. Local Memory 22
5.3. Usermode Synchronization 22
5.3.1. Busy-waiting synchronization 23
5.3.2. Process-switching synchronization 23
6. Building the Parallel Operating System 24
6.1. Data Structures 24
6.1.1. Queues 24
6.1.2. Hash Tables 24
6.2. Memory Allocation 25
6.3. Coordination Primitives 25

6.3.1. Busy-Waiting Synchronization 25
6.3.2. Process-Switching Synchronization 26
7. Machine Design 27
7.1. Network Design 28
7.1.1. Routing in an Omega-Network 28
7.1.2. Omega-Network Enhancements 29
7.1.2.1. Combining Loads and Stores 30
7.1.3. Implementing Fetch-and-add 30
7.1.3.1. Combining Fetch-and-adds 31
7.1.4. Network Switches 32
7.1.4.1. Packaging the Switch 34
7.1.4.2. Flow Control 35
7.1.4.3. The Forward Path Component 36
7.1.4.4. Return Path Component 37
7.1.5. Other Considerations 40
7.1.5.1. Pipelining Memory Requests 40
7.1.5.2. Limited Combining 41
7.1.5.3. Hashing or Interleaving Memory Addresses 41
7.2. The Network Interfaces 42
7.3. The Processors 42
7.4. Memory Hierarchy 42
7.4.1. Cache 43
7.4.2. Paging 44
7.5. Input/Output 46
7.6. Reflection and Refraction 47
7.7. Machine Packaging 47
8. Communication Network Performance 50
8.1. Performance Analysis 50
8.2. Network Simulations 54
9. Scientific Programming and Estimated Performance 55
10. Conclusion: Our Status and Future Plans 57
10.1. Prototype Hardware 57
10.2. Systems Software 57
10.3. Application Studies and Machine Simulation 58
10.4. Architecture Research 59
10.5. VLSI Design 59
References 61