A Hierarchically-Connected Multiprocessor

Jeffrey D. Martens
The Ohio State University
Department of Computer and Information Science
Columbus, Ohio 43210-1277
(614) 292-1932
martens@cis.ohio-state.edu

May 1, 1993
Revision of June 5, 1994

This work was presented as a poster at the 1994 Scalable High-Performance Computing Conference.

Abstract

A hierarchically-connected multiprocessor model is presented and compared to existing hierarchical shared-memory multiprocessor models. The key features of this architectural model include that a portion of the hierarchical memory is contained locally within each processing element, all memory is visible to every processing element, and each memory unit is local (but *not* private) to some processing element. Software and programming issues are explored through discussion of producer-consumer communication, barrier execution, and a hierarchical version of guided self-scheduling.

1 Introduction

Small-scale shared-memory multiprocessors have been built so that all addresses are equally accessible from every processor. Indeed, some authors have considered uniform accessibility to be a characteristic of shared memory[2]. The problem with this approach is two-fold: (1) all accesses are equally expensive, implying that all memory accesses in a large-scale machine will be expensive; (2) due to contention, access times cannot actually be uniform. To allow some accesses to be less expensive, shared-memory multiprocessors (SMMs) have been designed with local memories and/or caches; an example is the NYU Ultracomputer[10]. Thus on a multiprocessor with local memory, a program that can be written (or restructured) to exploit data locality will exhibit much less traffic within the interconnect, and thus be less adversely impacted by latency and contention. Multiprocessors in this category are termed non-uniform memory access (NUMA) machines.

In order to better take advantage of locality, hierarchical designs have arisen in which, for each processor, some memories are more efficiently accessible than others[15, 20, 21, 24]. These machines allow efficient sharing among various subsets of processors. In this paper, we consider any machine with three or more logical levels of memory (e.g., local, cluster, and global) to be hierarchical, and any machine with just one or two levels to be nonhierarchical.

Some large-scale SMMs have been designed as dance-hall machines, with the processors and the shared memories physically separate, and connected by a multistage interconnection network (MIN); the NYU Ultracomputer[10] falls into this category. Other large-scale SMMs have been designed with processing elements (PEs) connected to one another via a MIN, with each PE containing a portion of the shared memory as its local memory. Representatives of this class include the BBN Butterfly[5], the IBM RP3[1], and PASM[33]. For essentially nonhierarchical designs such as the above, researchers have convincingly shown that globally-accessible local memory (GALM) is a very useful feature in greatly reducing contention[9, 28, 27]; furthermore, as will be argued below, latency (in terms of number of switch stages traversed) can be reduced through the use of GALM.

Among hierarchical SMMs, some machines (e.g., Cedar[20], the THMM[15, 17, 25], the HMS[24]) have been designed without GALM, whereas in contrast some recent designs place a portion of each level of the hierarchical memory local to each PE[30, 35], the result being that the interconnect itself, rather than the memory *per se*, is hierarchical. In this paper the advantages of using a hierarchical interconnect—with all memory at the PEs themselves—rather than a hierarchical memory structure are explored. The result is that a portion of each level of the hierarchical memory resides in memory within each PE. We compare such an architecture with the more conventional approaches to hierarchical shared-memory multiprocessing, emphasizing programming and performance issues for such a multiprocessor.

Results in wormhole routing have lead some researchers to conclude that the distance messages travel is not an important factor in the performance of a multiprocessor, since, if the message length is large relative to the number of switch hops, the transit time of a message is dominated by the message length. This would seem to imply that locality (and, similarly, hierarchy) is not important, and if all processors were $\log p$ switch stages apart—or if, say, some processors were $\sqrt{p}$ switch hops apart—good performance could still be attained. However, in a SMM, many messages consist of only a single word. If the paths are a word wide, then the time to communicate a single word will be at best proportional to the number of hops the word must travel. Furthermore, many algorithms and some frequently-executed steps of algorithms require communication of small pieces of data; an example falling into this class is the barrier.

2 Architectural Models

It is typical to categorize parallel computers as either shared-memory multiprocessors (SMMs) or distributed-memory multicomputers (DMMs); SMMs communicate shared data through memory, whereas DMMs communicate via messages. A typical requirement for considering a machine to be an SMM is that the machine have a single address space which is accessible by every PE[2]. A characteristic typical of SMMs is that the time to access a single nonlocal memory location is much lower than the time to transmit a small message on a DMM; this startup time is typically masked in a DMM by packing many words into a single message. Similarly, vectorized reads and writes can be used to mask latency in SMMs.

On the other hand, it has been commonly argued that SMMs are not scalable due to contention for shared variables and latency of access to global memories. These problems have been addressed through various software techniques designed to split a variable among multiple memory units[34, 15] and to keep busy-waiting traffic to a minimum[28, 27, 9], and utilize local memories as much as possible, accessing global memory only when sharing is needed[3, 8]. The result is that, for maximum performance, it is necessary to borrow some distributed-memory techniques in order to program a SMM[23]. At the same time, there has been a complementary attempt to provide compiler support to allow the programmer to view a DMM as a SMM[7, 12, 19].

2.1 The HCM

We propose three logically-equivalent implementations of a hierarchically-connected multiprocessor. These three implementations are multiple networks, fast finishing, and fat trees, all of which are discussed below. By *logically-equivalent*, we mean that any two PEs are the same distance apart; this does not rule out cost or performance differences between the implementations.

2.1.1 Multiple Networks

A typical model for a NUMA SMM is a set of PEs—each containing a processor and a memory—connected by a multistage interconnection network (MIN) as in figure 1. The memory in this model

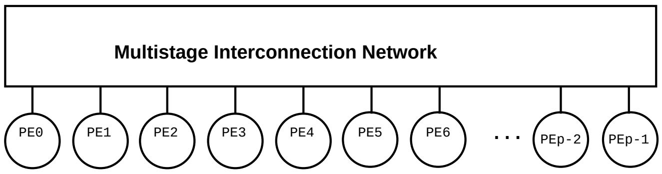

Figure 1: Typical NUMA system model.

can be thought of as a two-level hierarchy: local and nonlocal (or global). A third level can be added by adding a second set of networks as in figure 2. A four-level system (local, small cluster, large cluster, global) based on multiple networks is shown in figure 3.

2.1.2 Fast Finishing

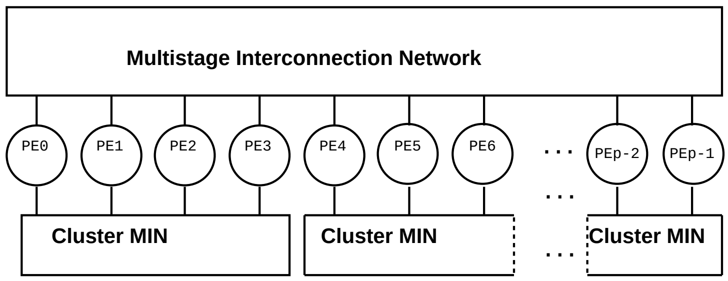

Figure 2: 3-level hierarchy.

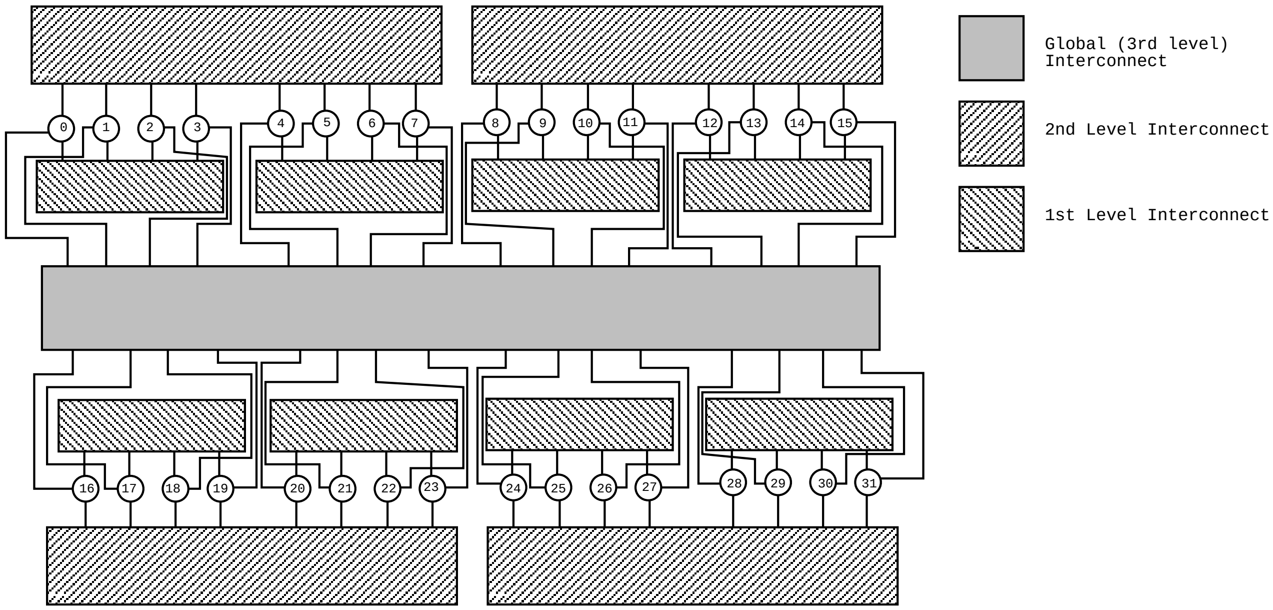

Figure 3: 32 PEs in a four level system. The circles are PEs, and the rectangles are switches.

Padmanabhan[30] points out that an address comparator can be added after one or more stages of the network to pull out packets destined for the memory unit in that column of the network; this reduces latency for packets that get to the proper column before the last switch stage (they need pass through fewer stages), and reduces contention in later stages of the interconnect since there are fewer messages traveling through those stages (this was also noted in [25, 26, 35]). Padmanabhan calls this use of comparators in the switch elements and connections from them directly to the memory units *fast finishing*. In figure 4 an eight PE system with one level of fast finishing is illustrated, resulting in a three-level system: global, clusters of size four, and local memories. Figure 5 shows a five level hierarchical system with local memory, three intermediate levels of fast

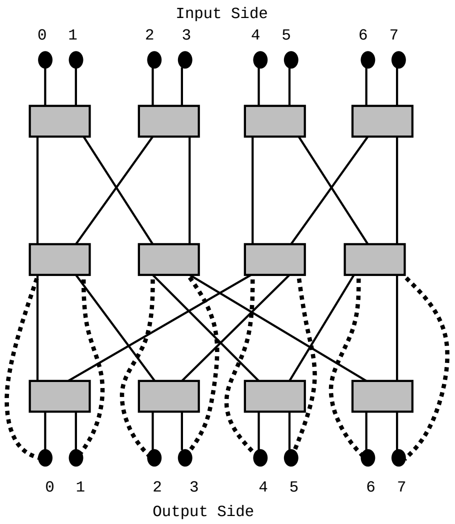

Figure 4: 8 PE indirect binary cube with fast finishing (shown with dotted lines) at the second switch stage.

finishing, and global memory. Inputs to the network are on the left, and outputs on the right. The fast finishing lines are bold. Note that input i and output i both connect to PE_i.

The hierarchically-connected multiprocessor (HCM) is based on the indirect binary cube network[29, 32]. A sixteen PE indirect cube is shown in figure 6.[1] The indirect cube network mentioned has p inputs and p outputs, for a total of p PEs. For any i in $0 \ldots p-1$, PE_i is connected to both input i and output i.

2.1.3 Tradeoffs Between Multiple Networks and Fast Finishing

Whether fast finishing or multiple networks are chosen is largely a matter of economics. Padmanabhan has shown that for a three-level system, multiple networks provide better performance. Fast finishing is likely to be less expensive, however. Due to expected traffic patterns, Wei and Levy suggest providing higher bandwidth low in the hierarchy, and decreased bandwidth higher[35]; one possible way of accomplishing this is by using multiple networks for the lower levels, but fast finishing for the higher levels.

Regardless of what combination of fast finishing and multiple networks is adopted, the resulting multiprocessor has a single, shared address space. The route by which a packet travels from a PE

[1] This system size is chosen for ease of illustration only. Multiple levels of hierarchy are most useful for large-scale machines with, say, hundreds or thousands of PEs.

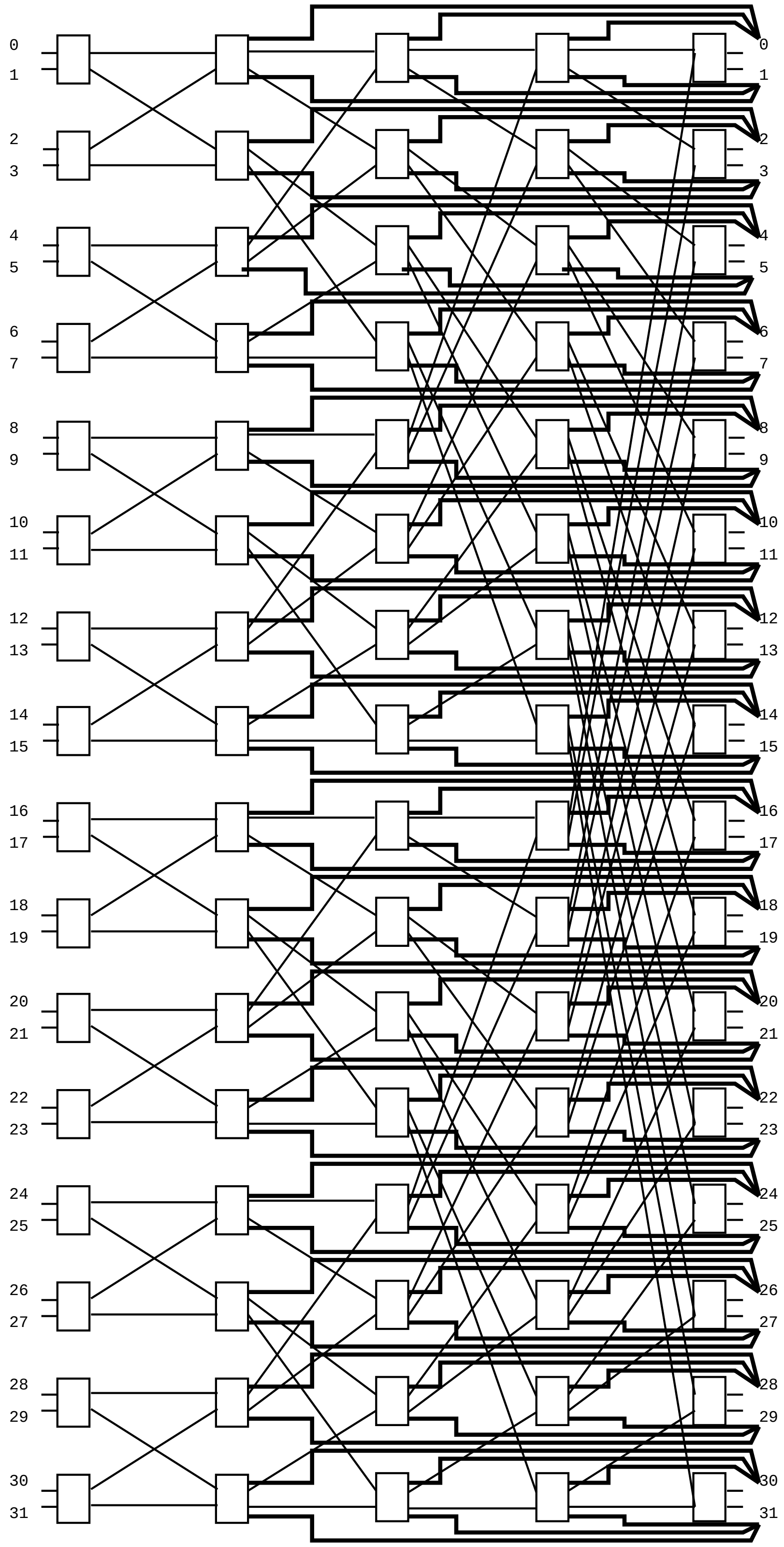

Figure 5: 5 level 32 PE system implemented using fast finishing.

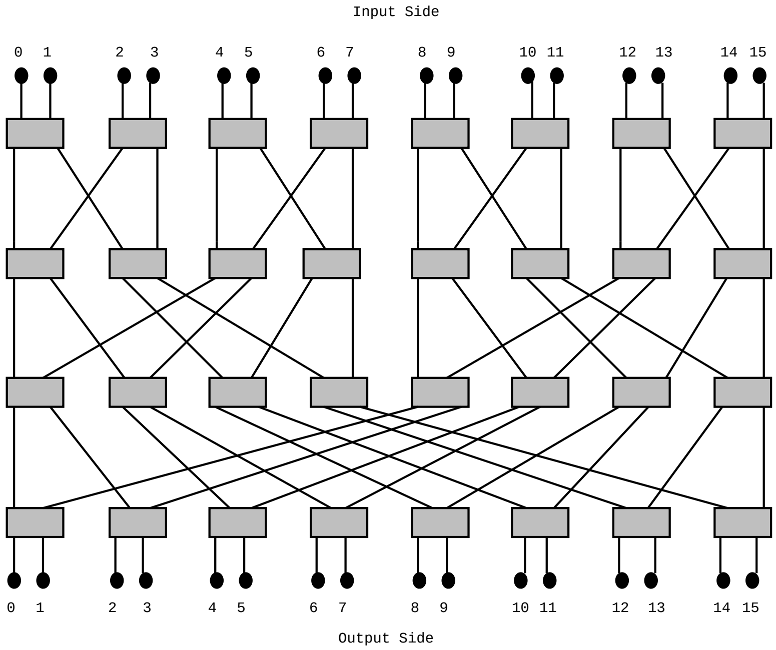

Figure 6: 16 PE indirect cube, $k = 2$.

to a memory, and thus the distance traveled, depends on the level of the interconnect at which the two are most closely connected. In Padmanabhan's fast finishing scheme, this determination of the shortest path is directly supported by the hardware. This can easily be supported in hardware in the multiple interconnect scheme as well. Adaptive routing schemes could also be used in case one or more networks become congested, though this has the drawback that the pipeline between a PE and a nonlocal memory would no longer be FIFO.

In practice it may be necessary to partition memory within a memory unit, or to provide multiple queues into a memory unit. If each memory unit includes multiple interleaved memories with pipelined fetch and store units, then interference between differing PEs attempting to access a memory at a particular point in time can be minimized. This is of course one motivation for limiting the number of levels of interconnect in the HCM; fewer levels implies a lower node degree for each PE, and fewer simultaneous requests arriving at each memory.

The optimal number of levels of hierarchy in an HCM is an open question. In a fast finishing-based system, using $k \times k$ switch elements, there will be a maximum of $\log_k p + 1$ levels. Using 8×8 switches, and letting $p = 4096$, would give us a maximum of five levels in the hierarchy: local memory, clusters of size eight, clusters of size 64, clusters of size 512, and global memory. In practice, one or more of these levels may be omitted. In the following we refer to an HCM with 2×2 switches and all $\log_2 p$ levels of hierarchy present as having the *full hierarchy.*

2.1.4 Fat Trees

Regardless of whether multiple networks or fast finishing is used to implement an HCM with full hierarchy, the interconnection network can be thought of as a k-ary tree, since for any level L in the hierarchy, there are k^L PEs in each cluster sharing data at that level. This suggests that a more explicitly tree-structured network may bear investigation. The fat tree is a network which is tree-structured with processors as leaves and switches as internal nodes[11, 22]. The reason the tree is termed "fat" is that the bandwidth between nodes is higher closer to the root than near the PEs—the tree does not necessarily grow narrower as the root is approached. A fat tree in which bandwidth is constant at every level of the hierarchy may be seen in figure 7.

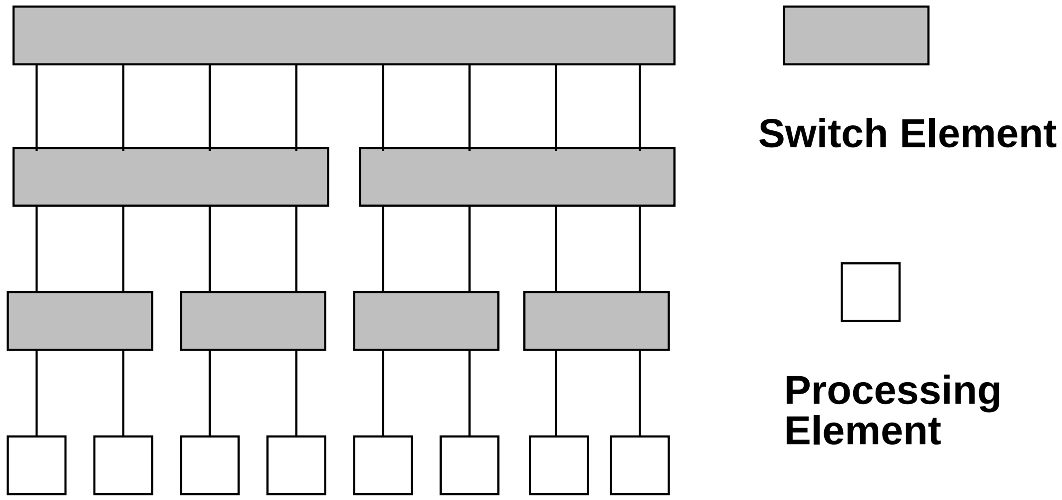

Figure 7: A fat tree.

A shared-memory fat tree is logically equivalent to the HCM models presented above, except that the fat tree necessarily has the full hierarchy property. Thus, this is a third possible HCM implementation. A disadvantage of the fat tree is that any two processors are exactly twice as many network hops apart as they would be in the prior HCM models. All three HCM models share the property that a portion of each level of the hierarchical memory is locally accessible to each PE.

2.2 The THMM

The tree-hierarchical memory multiprocessor (THMM) is a SMM based on an indirect binary cube interconnect in which the PEs and memory units are distinct[2] (though there are also local memories within the PEs). One possible implementation of the THMM places an additional memory within each switch in the network; for convenience we will refer to this organization as the THMM itself. One characteristic of the indirect binary cube network is that from any memory or switch, a tree can be formed with PEs as the leaves. Thus, the THMM can be considered to be tree structured, with PEs as leaves and memories as interior nodes (see figure 8). The primary difference between the HCM and the THMM is that a portion of each level of the hierarchy is locally-accessible by each PE in the HCM, whereas in the THMM the hierarchy is mostly separate from the PEs.

[2]That the PEs and global memory units are distinct, i.e., that the network does not wrap around, has not been considered a necessary aspect of the THMM. For the sake of the current paper, we assume that the the global memories are not contained within corresponding PEs.

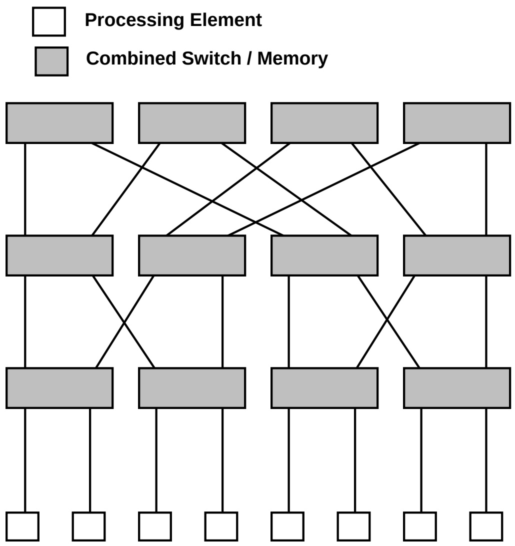

Figure 8: Eight PE THMM, forward network only.

2.3 Cedar

Cedar[20] is a hierarchical SMM with a three-level hierarchy: local cache, cluster, and global. Within a cluster there is hardware for fast synchronization; across clusters, synchronization is handled through global memory. None of the global memory units is local to any PE[20].

Although the Cedar prototype has only four clusters of eight PEs each, it was conceived with scalability in mind. In this paper, *Cedar* is used to refer to a large-scale SMM structured similarly to the Cedar prototype. The number of clusters and the number of PEs per cluster are not taken to be fixed quantities herein.

An architecture that is in many respects similar to Cedar is the three-level Hierarchical Memory Structure (HMS) of Mabbs and Forward[24]. The three levels in this HMS are local memory, cluster memory, and global memory. The local memory is private to the associated PE, and none of the global or cluster memory is local to any PE. Thus, in the following discussion most comments regarding Cedar also apply to the HMS.

2.4 Immediate Consequences of Architectural Models

There are two immediate consequences of placing a portion of each level of the hierarchical memory local to each PE, as in the HCM, as opposed to the other models discussed. These are low-contention spin waiting and reduced latency of naively partitioned data sets.

2.4.1 Spin Waiting

Mellor-Crummey and Scott have demonstrated the utility of globally-accessible local memory (GALM) for synchronization and communication[27, 28]. The main idea is that if the software can be structured so that waiting PEs check the status of local variables only, then no busy-wait traffic arises. Several algorithms are given for accomplishing this in [27, 28]; these algorithms assume that the multiprocessor supports fetch-and-op instructions and that there is either GALM or coherent caches; for the HCM, GALM was chosen.

2.4.2 Performance with Poor or Naive Partitions

One advantage that SMMs have over DMMs is that if the program is naively partitioned, data can still be directly accessed, though possibly at a penalty of greater access time than if the data had been partitioned optimally[6]. This is true for either Cedar or the THMM, since it is always possible to place all data at the global level. This is also true for the HCM, but it is important to observe that in the HCM a poor or naive partition will generally allow at least some data to be closer than the global level. Thus, unlike Cedar or the THMM, the HCM enjoys the benefits of hierarchy with even the most naive partitioning. In all three machines, the benefits increase as partitioning improves.

3 Programming the HCM

Placing a portion of each level of the hierarchical memory in each PE improves multiprocessor performance in producer-consumer communication, the execution of synchronization barriers, and in dynamically scheduled programs. The reasons for this are outlined in this section.

3.1 Producer-Consumer Communication

Jordan[18] divides producer-consumer communication into two categories: consumer-initiated, and producer-initiated.

3.1.1 Producer-Initiated Communication

In producer-initiated communication, the producer "knows" the identity of the consumer, and thus writes into the location where the consumer expects to find the produced item. This is handled similarly in all three architectures under consideration:

- In the THMM, the item is written to an agreed upon location in the root of the smallest subtree containing both the producer and the consumer.
- In Cedar, the item is written to a cluster memory if the producer and the consumer are both known to be in the same cluster. Otherwise, the item is written to a location in the global memory.
- In the HCM, the item is written to a location in the memory local to the consumer. The reduction of distance between the producer and the consumer is similar to that in the THMM (they are equivalent if the HCM has a full hierarchy).

The main difference between the three architectures in producer-initiated communication is that the consumer in the THMM or Cedar must obtain the item from a nonlocal variable, whereas on the HCM the consumer obtains the item locally. On the THMM, the consumer spins on a location in the hierarchy, waiting for the producer to write into the location. In the worst case, multiple producer-consumer pairs can cause considerable contention as the consumers spin on their various not-yet-produced items.[3] Jayasimha has pointed out that this situation can be greatly improved by incorporating a signalling mechanism by which the memory unit informs the consumer when the desired item arrives[16]. The produced item still must travel from the producer to the memory, and then from the memory to the consumer, rather than directly from the producer to the consumer, increasing the latency relative to what would be expected on the HCM.

On Cedar, if the producer and consumer are in different clusters, considerable busy-wait traffic can arise (within a cluster consumers spin on local cache lines). Furthermore, the produced item must travel from the producer to the memory, and then finally to the consumer.

Thus a major advantage of the HCM over Cedar or the THMM in producer-initiated communication is in the distance the produced item must travel. A second advantage is in reduced traffic: on the HCM, the only traffic in the network caused by one producer-consumer pair is the producer writing into the consumer's local memory. On the other two machines, there is approximately three times as much traffic: the producer's write, the consumer's read request, and the memory unit satisfying the read.

3.1.2 Consumer-Initiated Communication

In consumer-initiated communication, the producer does not necessarily have any "knowledge" of where the consumer resides. The consumer, on the other hand, must know where the producer will deposit the item. On Cedar or the THMM, the producer must place the item in the global level of the hierarchy, in time $\Omega(\log p)$, where the lower bound on time is given by the number of switch stages the write must traverse. On the HCM, the producer can place the item in its own local memory, in constant time. The consumer may or may not be in a cluster including the producer; if the producer and consumer are in a common cluster, then the consumer's access requires fewer than $\log_k p$ network hops to reach the item; otherwise, the situation is the same as in Cedar or the THMM: the consumer's request must traverse the entire interconnect. So, the HCM's advantage is two-fold: (1) the produced value takes $\theta(1)$ time to appear in the location at which the consumer waits, rather than the $\Omega(\log p)$ time required for the THMM or Cedar, and (2) the consumer's read request may not need to traverse the entire interconnect.

3.2 Synchronization Barriers

The barrier is a fundamental synchronization operation in which all processes wait until every process is known to have reached a certain point in program execution (the barrier) before continuing. For simplicity, we assume that there is exactly one process per PE, but multiple processes per PE are easily handled. A barrier can be implemented as a single globally-accessible counter; each PE increments this counter and waits for the value of the counter to reach p, at which time the barrier is said to be complete, and execution continues at the point immediately following the barrier. This scheme, though simple, suffers greatly from contention: each PE must atomically modify the counter and then spin-wait. The contention due to the spin-waiting can be reduced

[3] The contention arises if the paths between the various consumers and the memory units the consumers are spinning at are not disjoint. In a structured case, it is often possible to arrange the producers and consumers so that there is little or no contention[15, 17, 25].

through backoff techniques, in which PEs pause between accesses of the counter[14], or the amount of contention can be reduced by splitting the counter among different memory units and combining their values in a tree-like fashion[4, 13, 34]. This tree barrier is illustrated in figure 9. In figure 9,

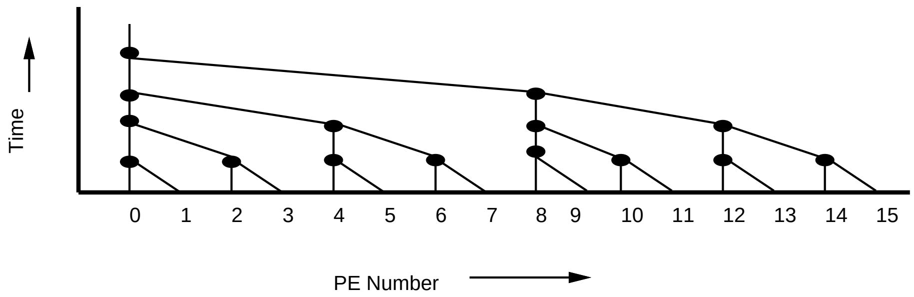

Figure 9: Sixteen PE 2-ary tree barrier.

each dark circle represents a synchronization point. Time progresses as we move from the bottom to the top of the figure, and the number of levels in the tree is $\log_2 p$. So, along the critical path there are $\log_2 p$ synchronizations. At each synchronization point, the lower-numbered of the two synchronizing PEs continues to the next level.[4] The barrier proceeds through an *arrival* (or *walk-in*) phase which ascends the tree as just described. This is followed by a *notification* (or *walk-out*) phase in which the tree is traversed from top to bottom, until each PE has been informed that the barrier is complete.

In the HCM each node in the barrier tree is placed at one of the synchronizing PEs. Assume this is the lower-numbered of the two PEs. Consider PE_0; PE_0 waits for PE_1 to modify a variable local to PE_0; after receiving PE_1's arrival, PE_0 waits for PE_2's arrival. In the meantime, PE_1 waits for notification of the barrier completion from PE_0; this notification will take the form of PE_0 setting a flag within PE_1's local memory; the other PEs behave similarly. Thus the barrier moves up the tree, and then back down, with no PE ever spinning on a variable within another PE's memory, and so there is little contention. If the HCM uses 2×2 switches and has a full hierarchy, the distance a message must travel in the lowest level of the barrier tree is just one hop, two hops in the second, and so forth. It is only at the root of the tree, when PE_0 is synchronizing with $PE_{\lfloor p/2 \rfloor}$, that the write requests must traverse a full $\log_2 p$ stages of the interconnect. The total delay due to switch latency is $\theta(\log_2^2 p)$.

The THMM barrier is similar, except instead of writing into a memory unit local to the lower numbered of a synchronizing pair, writes occur into a memory location in a memory unit at the root of the smallest subtree containing both PEs. The contention is similarly minimal—though the THMM PEs will spin on nonlocal locations, the tree is arranged so that there is only one PE spinning within a subtree at a time. The overall impact of latency on the THMM will, all other things being equal, be twice that of the HCM, since accesses must travel twice as far.

In Cedar, the tree barrier can be simplified: synchronization within a cluster proceeds in roughly constant time, so the height of the tree is $\log_2 C$, where C is the number of clusters. Typically cluster size will be relatively small, so that this does not greatly decrease the number of levels in the

[4]There are other ways of implementing a tree barrier—this is among the simplest, however. For example, the PE proceeding to the next level could be the latest arriving at a synchronization point. For efficiency, 4-ary or wider trees are often chosen rather than binary trees.

barrier tree. Contention is higher than in the other two models due to spinning on global locations; and overall latency is higher in Cedar than in either other machine since the $\log_2 C$ levels of the tree are all in global memory.

3.3 A Hierarchical Version of GSS for the HCM

For good load balancing, parallel (or `doall`) loops are often scheduled dynamically (i.e., at runtime). In order to do this efficiently, a scheme that doesn't require each iteration to be dispatched individually from a central location should be used. One method for grouping iterations, guided self-scheduling (GSS), was given by Polychronopoulos and Kuck[31]. In this method, a central counter is used, but each PE allocates itself blocks of iterations rather than single iterations. The number of iterations allocated by a PE needing work to do is the number of remaining iterations divided by the number of PEs; thus the number of allocations is logarithmic (rather than linear) in the number of iterations.

A similar scheme can be used on the HCM. Instead of having a central counter, however, it is useful to have an iteration counter local to each PE. Each PE's counter is initially allocated a set of n/p of the n loop iterations, and each PE allocates itself iterations out of its local set until they are exhausted. Then the PE moves up a level in the hierarchy, allocating itself iterations out of other PEs' sets. Whenever the iterations of all the PEs at one level are exhausted, a PE looking for work moves up to the next level. Thus, most of the executions should be allocated out of local memory and from sets in PEs' memories in the lower levels of the hierarchy. Pseudocode for this is given in figure 10; this pseudocode assumes a full hierarchy and 2×2 switches.

Some aspects of the algorithm in figure 10 bear discussion. Each PE executes procedure `HGSS` in parallel, with the parameter m indicating the PE number. The call to `allocate` locks the selected PE iteration set; these locks can be written to avoid nonlocal spinning[27, 28], so lock contention is not expected to be a serious problem. `allocate` can return an empty iteration range (indicated by $lo > hi$); assuming Pascal-like `for` loop semantics, `HGSS` executes zero iterations in such cases. `HGSS` can determine whether a PE is a leaf PE in the barrier in the conditional by determining whether $2m \geq p$; if so, PE_m is a leaf. Finally, the iteration range in the inner `for` loop is the set $\{m \mid \mathrm{PE}_m$ is accessible at this level but not at lower levels$\}$; to avoid contention, this should be implemented so that different PEs do not search other PEs' memories in the same order.

Unlike the original version of GSS, each allocation takes half the remaining iterations rather than $\frac{1}{p}$ of them. This is because the iterations have already been partitioned across p memory units.

$O(\log n/p + p)$ allocations are made from each PE's local set, the additional p term arising due to the fact that in worst case every processor checks every other processor's set. To make this worst case unlikely, a barrier is embedded into the algorithm: whenever a PE's local set is exhausted, and the PE's children have informed it that their local sets are exhausted, than the PE informs its parent that the sets in its subtree are exhausted. Eventually, this information propagates to the root of the synchronization tree, at which time PEs begin notifying their children. Thus there should little effort lost in searching empty PEs' sets after there are no iterations remaining in any PE. This also means that there are two exits from `HGSS`: the procedure returns when all PE sets have been exhausted (after the nested loop runs to completion), or when `notify[m]` is true.

No PE checks an empty partition more than once, and so contention for any given partition is no higher than at the global run queue in Polychronopoulos' original GSS. At each intermediate step, the distance between the PE and the memory is less than the full interconnect, so overall

```
shared data
        q:  array [ 0.. p-1 ] of integer
                q[m] is placed on PE[m]
                q[m] is initially mn/p
        notify:  array [ 0.. p-1 ] of boolean
                notify[m] is placed on PE[m]
                notify[m] is initially false
        arrive:  array [ 0 .. p ] of boolean
                arrive[m] is placed on PE[(m-1) div 2]
                arrive[0 .. p-1] is initially false, arrive[p] is initially true

function allocate[m] ==
        local integer remain, lo, hi -- allocated in stack of calling PE

        lock q[m]
        remain := (m+1)n/p - q[m]
        lo := q[m]
        hi := q[m] + ceiling(remain / 2) - 1
        q[m] := max(q[m], q[m] + hi - lo + 1)
        unlock q[m]
        return lo, hi
end allocate

procedure HGSS(m) ==
        local boolean passed := false
        local integer lo, hi, i, level

        while local iteration set is nonempty do
                lo, hi := allocate[m]
                for i := lo to hi do iteration i od
        od
        if (leaf PE) then
                arrive[m] := true
                passed := true
        fi
        for level := 1 to lg p do
                for source in {m|m is a PE accessible at this level but
                                not lower levels} do
                        if arrive[2m+1] and arrive[2m+2] and not passed then
                                arrive[m], passed := true, true
                                if m = 0 then notify[0] := true fi
                        fi
                        if notify[m] then
                                notify[2m + 1], notify[2m + 2] := true, true
                                return
                        fi
                        lo, hi := allocate(source)
                        while lo <= hi do
                                for i := lo to hi do iteration i od
                                lo, hi := allocate(source)
                        od
                od
        od
end HGSS
```


Figure 10: Hierarchical GSS.

latency should be lower than if a global counter were used.

4 Conclusion

A hierarchically-connected multiprocessor has been presented, and useful features of the hierarchy have been discussed. In particular, it has been shown that placing a portion of every level of the hierarchical memory local to each PE is useful in reducing both latency and contention. This memory organization can be brought about through the use of multiple networks, through the use of fast finishing, or through the use of a shared-memory fat tree. An HCM based on multiple MINs or fast finishing is not much more expensive than a typical NUMA multiprocessor, as long as the number of levels of hierarchy is kept relatively low. An HCM based on a shared-memory fat tree is of approximately the same cost as a typical MIN-based NUMA multiprocessor.

References

[1] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, New York, 1989.

[2] Gordon Bell. Ultracomputers: a Teraflop Before its Time. *Communications of the ACM*, 35(8):26–48, August 1992.

[3] Randy Bramley and J. Bordner. Sequential Optimization and Data Distribution for ARC2D on the Cedar Hierarchical Multiprocessor. Technical Report 1128, Center of Supercomputing Research and Development, University of Illinois, September 1991.

[4] Eugene D. Brooks III. The Butterfly Barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[5] W. J. Downey III. *Inside the GP1000.* BBN Advanced Computers, Inc., Cambridge, MA, 1988.

[6] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua. Restructuring FORTRAN Programs for Cedar. In *International Conference on Parallel Processing*, volume I, pages 57–66, August 1991.

[7] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D Language Specification. Technical Report COMP TR90-141, Department of Computer Science, Rice University, December 1990. Revised April, 1991.

[8] Kyle Gallivan, William Jalby, and Dennis Gannon. On the Problem of Optimizing Data Transfers for Complex Memory Systems. In *International Conference on Supercomputing*, 1988.

[9] Lal George. A Scheduling Strategy for Shared Memory Multiprocessors. In *International Conference on Parallel Processing*, volume I, pages 67–71, 1990.

[10] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[11] R. I. Greenberg and C. E. Leiserson. Randomized Routing in Fat-Trees. In *IEEE 26th Annual Symposium on the Foundations of Computer Science*, 1985.

[12] Andrew S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing With Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[13] Debra Hensgen, Raphael Finkel, and Udi Manber. Two Algorithms for Barrier Synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[14] D. N. Jayasimha. Parallel Access to Synchronization Variables. In *International Conference on Parallel Processing*, pages 97–100, August 1987.

[15] D. N. Jayasimha. Partially Shared Variables and Hierarchical Shared Memory Multiprocessor Architectures. In *11th Annual IEEE International Conference on Computers and Communications*, pages 63–71, April 1992.

[16] D. N. Jayasimha. Personal communication, April 1993.

[17] D. N. Jayasimha and J. D. Martens. Some Architectural and Compilation Issues in the Design of Hierarchical Shared Memory Multiprocessors. In *International Parallel Processing Symposium*, pages 567–572, March 1992.

[18] Harry F. Jordan. Shared Versus Distributed Memory Multiprocessors. Technical Report 91-7, ICASE, January 1991.

[19] Charles Koelbel and Piyush Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[20] J. Konicek et al. The Organization of the Cedar System. In *International Conference on Parallel Processing*, volume I, pages 49–56, August 1991.

[21] James T. Kuehn and Burton J. Smith. The Horizon Supercomputing System: Architecture and Software. In *Supercomputing '88*, pages 28–34, 1988.

[22] Charles E. Leiserson. Fat-Trees: Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computing*, C-34:892–901, October 1985.

[23] Calvin Lin and Larry Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. In *International Conference on Parallel Processing*, volume II, pages 163–170, August 1990.

[24] Stephen A. Mabbs and Kevin E. Forward. Optimizing the Communication Architecture of a Hierarchical Parallel Processor. In *International Conference on Parallel Processing*, volume I, pages 516–520, August 1990.

[25] J. D. Martens and D. N. Jayasimha. A Tree-Structured Hierarchical Memory Multiprocessor. Technical Report 61, Department of Computer and Information Science, The Ohio State University, December 1989. Revised May 1990. An extended abstract appeared in *International Conference on Parallel Processing*, volume I, pages 561–562, August 1990.

[26] J. D. Martens and D. N. Jayasimha. Performance of a Tree-Structured Hierarchical Memory Multiprocessor. Technical Report 7, Department of Computer and Information Science, The Ohio State University, January 1992.

[27] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[28] John M. Mellor-Crummey and Michael L. Scott. Synchronization without Contention. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, April 1991.

[29] Krishnan Padmanabhan. Cube Structures for Multiprocessors. *Communications of the ACM*, 33(1):43–52, January 1990.

[30] Krishnan Padmanabhan. Efficient Architectures for Data Access in a Shared Memory Hierarchy. *Journal of Parallel and Distributed Computing*, 11:314–327, April 1991.

[31] C. D. Polychronopoulos and D. J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36:1425–1439, December 1987.

[32] Howard J. Siegel, Wayne G. Nation, Clyde P. Kruskal, and Leonard M. Napolitano. Using the Multistage Cube Network Topology in Parallel Supercomputers. In *Interconnection Networks for Large-Scale Parallel Processing* by Howard J. Siegel, McGraw-Hill Publishing Company, New York, 1990. Pages 313–364.

[33] Howard Jay Siegel, Wayne G. Nation, and Mark D. Allemang. The Organization of the PASM Reconfigurable Parallel Processing System. In *First Annual OSU Workshop on Parallel Computing*, pages 1–12. The Department of Computer and Information Science, Ohio State University, 1990.

[34] Peiyi Tang and Pen-Chung Yew. Software Combining Algorithms for Distributing Hot-Spot Addressing. *Journal of Parallel and Distributed Computing*, 10, 1990.

[35] Sizheng Wei and Saul Levy. Efficient Hierarchical Interconnection for Multiprocessor Systems. In *Supercomputing '92*, pages 708–717, 1992.