A FINE-GRAIN PARALLEL ARCHITECTURE BASED ON BARRIER SYNCHRONIZATION

H. G. Dietz, R. Hoare, and T. Mattox

Purdue University, School of Electrical and Computer Engineering
West Lafayette, IN 47907-1285
`hankd@ecn.purdue.edu`
`http://garage.ecn.purdue.edu/~papers`

Although barrier synchronization has long been considered a useful construct for parallel programming, it has generally been either layered on top of a communication system or used as a completely independent mechanism. Instead, we propose that all communication be made a side-effect of barrier synchronization. This is done by extending the barrier synchronization unit to collect a datum from each processor, compute an aggregate function, and return the corresponding result to each processor.

This paper describes a scalable prototype implementation of PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization). Despite the fact that the prototype is implemented as very simple TTL hardware connecting conventional workstations, measured performance on fine-grain parallel communication operations is far superior to that obtained using conventional workstation networks. It is comparable to the performance of commercially available supercomputers.

1. Introduction

To obtain good speed-up of most programs by executing portions of the code in parallel, the parallel hardware must provide more than just multiple processors. A wide range of parallel architectures have been proposed, each focusing on a different aspect. Rather than reviewing all these architectures, we found it more productive to go back to the basic problems which must be solved, and to derive the architectural model directly from these needs.

In his popular textbook on high-performance computer architecture [7], Stone provides a simple list of the primary factors that can lead to poor performance. The five issues he cites, and our interpretations of them are:

- **Delays introduced by interprocessor communications.** Thus, low latency is at least as important as high bandwidth.
- **Overhead in synchronizing the work of one processor with another.** Synchronization of two, or more, processors must be a direct hardware function, with minimal latency.
- **Lost efficiency when one or more processors run out of tasks.** It must be possible to efficiently sample the global state of the machine and to re-allocate work based on this sampling so that such imbalance is avoided.
- **Lost efficiency due to wasted effort by one or more processors.** Again, we see this problem as centering on the need to have low latency access to a sampling of global state, since that alone can allow processors to recognize and avoid redundant or unnecessary computations.
- **Processing costs for controlling the system and scheduling operations.** To minimize runtime scheduling cost is to maximize the fraction of this work that can be accomplished at compile time. Thus, it is critical to provide a dynamic runtime environment in which properties required by the compiler's static schedule can be directly enforced. The basic mechanisms to support this are low latency barrier synchronization to enforce timing constraints and operations that allow the appropriate static schedule to be selected based on the system's dynamic global state.

From this, we observe that barrier synchronization augmented by a mechanism for sampling global state can potentially solve all these problems.

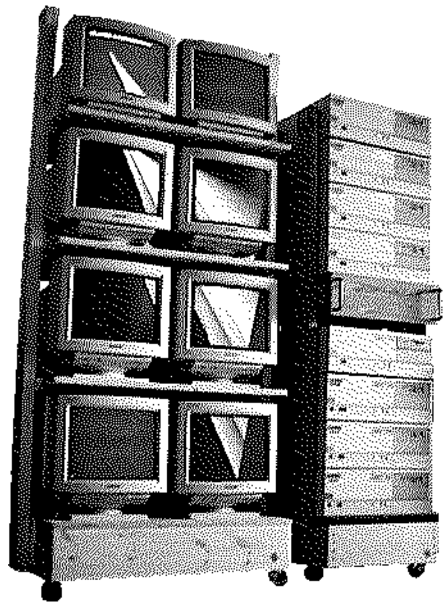

Figure 1: Eight Processor TTL_PAPERS 951201 Cluster

In section 2, the structure of a scalable prototype implementation of the TTL_PAPERS architecture (cluster shown in Figure 1) is described. The performance of this prototype system is summarized in section 3. In conclusion, section 4 reviews the advantages of this new architecture.

2. TTL_PAPERS 951201 Architecture

The TTL_PAPERS 951201 architecture is comprised of four subsystems: synchronization, parallel signaling, data communication and status display. The data communication subsystem is a data network that allows communication after a barrier has been reached. The parallel signaling subsystem can be used by a processor to asynchronously signal the other processors and can be used in much the same way as an interrupt. The status display subsystem allows the user to see when a processor is waiting at a barrier, performing computation, being controlled by the operating system, or inactive.

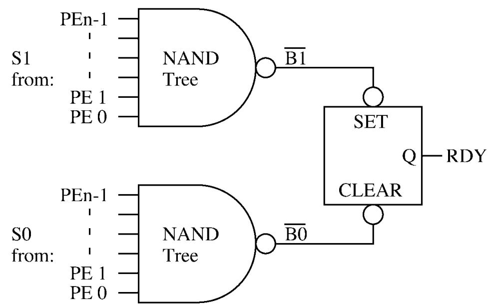

Figure 2: Static Barrier Mechanism of TTL_PAPERS

2.1. Synchronization Subsystem

Hardware barrier synchronization was first proposed in a paper by Harry Jordon [4], and has since become a popular mechanism for coordination of MIMD parallel processes. A barrier synchronization is accomplished by processors executing a `wait` operation that does not terminate until sometime after all processors have signaled that they are waiting. Experiments with the PASM (PArtitionable Simd Mimd) prototype's barrier mechanism [6] led us to significantly extend the concept of barrier synchronization. The TTL_PAPERS architecture uses a variation on the SBM (Static Barrier MIMD) design of [5] to allow a more direct prototype implementation. The primary difference between the previously published SBM and the TTL_PAPERS mechanism is that there are two barrier trees rather than one (see Figure 2). The reason is simply that the published SBM silently assumed that the barrier hardware would be reset between barriers. In contrast, the use of two trees allows the hardware for one tree to be reset as a side-effect of the other tree being used.

Typically, a barrier synchronization consists of four cycles: request a barrier, observe that the barrier is done, reset the barrier unit, and observe that the barrier unit has been reset. Barrier synchronization is typically implemented using a barrier tree which can be viewed as a large AND gate, where each processor supplies a barrier request signal, S1, to an input of the AND gate and the output which is the barrier done signal, B1. This can be seen as the upper NAND Tree of Figure 2. The barrier done signal, B1, transitions to a logic high only when all processors have set their barrier request signal, S1, high. This is essentially the same as the barrier/eureka network in a Cray T3D [2]. Each processor can then test the resulting signal to determine when all processors have reached the barrier. However, a single barrier tree has to be reset before it can be used again. This presents two serious problems:

- None of the processors can clear their S1 signal until all of the processors have observed the B1 (barrier done) signal. Otherwise, a barrier request can go low before all the processors have seen the barrier done signal in its high state. Thus, some processors can get stuck waiting at a barrier that has already been completed.
- Before a processor can set its barrier request signal high again at the next barrier, all other processors must have already cleared their S1 signal from the first barrier. Otherwise, a processor with little or no work to do between the first and second barriers might clear and then set its S1 signal again while other processors still have their S1 signals high due to the first barrier. This race condition can cause the B1 signal to erroneously go high, and if observed by any processor, would be seen as "the second barrier is done," which is incorrect.

The first problem can be solved by using a one bit register that is set to one when the B1 signal goes high, as shown in Fig. 2. Then all processors can asynchronously observe that the barrier is complete by reading the output signal, RDY, of the register.

The second problem is solved by literally having a second barrier tree that indicates when all processors have cleared their S1 signals. One way to do this is to have a second signal, S0, from each processor that is set when that processor has cleared its S1 signal. Thus the second barrier tree generates a signal, B0, when all processors have cleared their S1 signals. A direct way of utilizing this B0 signal would be to clear the RDY register when B0 goes high. The end result of all this is that as each barrier in a sequence is reached, the RDY signal toggles. Since the transition of the RDY from a high to a low is a valid synchronization at a barrier, the act of resetting the B1 barrier tree can itself be used as a barrier.

An additional benefit to having two synchronization signals, S1 and S0, is the ability for processors to remove themselves from the barrier group. This allows the machine to partition into two sets: those processors using this barrier unit and those who are not. This is accomplished by a disabled processor setting both its S1 and S0 signals high. Both the barrier trees would then effectively ignore the state of the disabled processors.

This architecture scales to arbitrarily large systems, since the electrical propagation delay of an AND tree is O(log N), with O(2N) components in the tree. Note that the magnitude of the delay time is measured in nanoseconds for typical circuit implementations.

2.2. Communication Subsystem

The TTL_PAPERS library provides a rich array of aggregate communication operations, most with total process-to-process latency in the tens of microseconds. This is accomplished without packetizing, routing, or switching by constructing more complex aggregate functions using a 4-bit wide NAND tree. Clearly, operations like bitwise AND or OR are directly implemented by the NAND tree, but so are many other aggregates. For example, a broadcast is implemented by one processor contributing the complement of its 4-bit datum while all others hold their outputs high. NAND hardware also implements a variety of voting operations. A more detailed discussion of the communication library can be found in [3].

2.3. Parallel Signaling Subsystem

The parallel signaling subsystem, shown in Fig. 3, is similar to that of the barrier unit. At any time, a processor can asynchronously set the parallel signal, IRQ, high. Each of the other processors is then required to acknowledge that it has seen the INT signal by setting its IACK signal high. When all

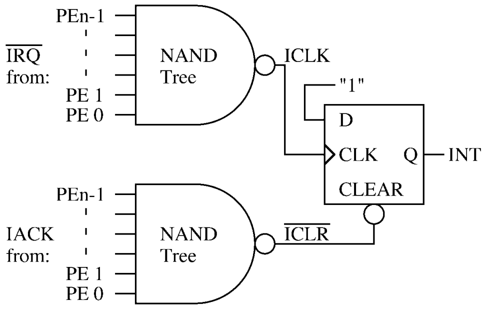

Figure 3: Asynchronous Parallel Signaling Subsystem

processors have acknowledged the parallel INT signal, it is cleared. Although this mechanism is implemented very differently, in function it is very similar to an interrupt and can be used as an interrupt mechanism.

PAPERS uses an alternative approach to interrupts, called parallel signaling, that does not require any context switches and can still be used by the programmer in the same way as interrupts. When a parallel signal is issued, the PAPERS unit registers the request and does not allow any barriers to be completed until all of the processors have seen the parallel signal. This is done by using the barrier library to detect the parallel signal. After sending a barrier request, the processor waits for the barrier acknowledge to be completed. During this waiting time the processor checks to see if the parallel signal has been set. If it has been set, the processor acknowledges that it has seen the signal, executes the signaling routine, and then resumes waiting for the barrier. This does not use any operating system routines and spends the idle time waiting at a barrier performing useful work. Thus the efficiency of parallel signaling increases as the granularity becomes smaller.

2.4. Hardware/Software Interface

The PAPERS architecture was designed for extremely low latency and thus, the hardware/software interface was optimized. The operating system was avoided because of context switching time and thus, the software libraries communicate with the hardware directly through I/O hardware registers. The TTL_PAPERS was designed so that a single I/O write could request a barrier and a single I/O read could be used to detect when the barrier is done. At the same time the barrier completion is detected, data is also received. This allows data communication to be added to the architecture without incurring a large amount of overhead.

2.5. Scalability

It can be seen in Figures 1 and 4 that a single TTL_PAPERS 951201 unit can support 8 processors. The TTL_PAPERS 951201 also has four additional inputs as well as four additional outputs. By using these inputs and outputs, TTL_PAPERS 951201 can scale in a tree like fashion. Figure 5 shows how a 168 processor system can be constructed with twenty-one TTL_PAPERS 951201 units.

When a barrier is requested, the both signals and data travel up the tree to the root node, which, upon completion of the barrier synchronization, broadcasts the result back down the

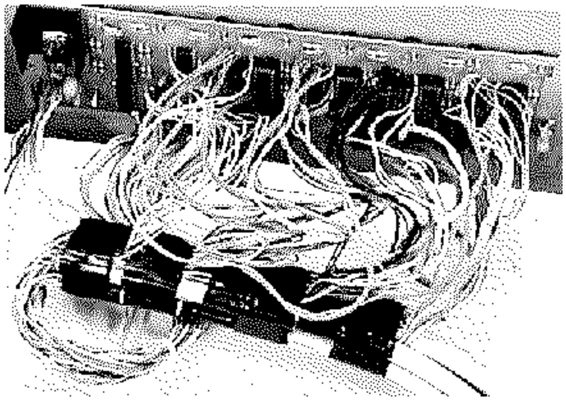

Figure 4: Internal wiring of an 8 processor TTL_PAPERS 951201

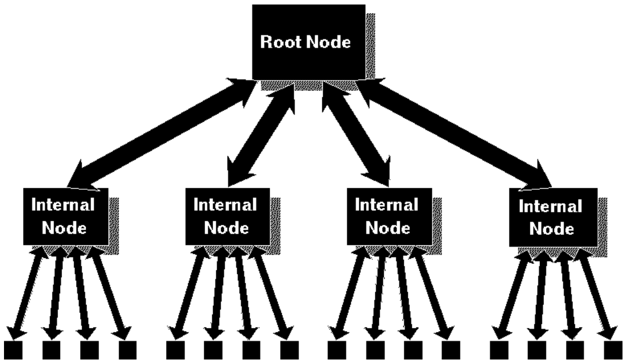

Figure 5: 168 Processor TTL_PAPERS 951201 Tree Structure

tree. While the result is traveling down the tree, it is also sent out to the processors that are connected to the individual PAPERS units. In this way, TTL_PAPERS 951201 can be scaled to over two thousand processors with a tree height of only five.

3. Performance

The performance of the TTL_PAPERS design is limited primarily by the fact that it is connected via the parallel port of each machine within the cluster. Although the latest EPP and ECP "enhanced" parallel ports could improve performance by an order of magnitude, the TTL_PAPERS design was made compatible with the more universally available "standard" parallel ports. Thus, the design uses software handshaking and data transmission is limited to a 4-bit path.

3.1. Performance Model

Each reference to a standard parallel port register takes between 900ns and 5us, with 1us-2us typical. In comparison, the propagation delay caused by cables is negligible, with a worst-case of approximately 100ns. Likewise, the worst-case propagation delay through the logic of the TTL_PAPERS unit is merely 78ns when using 74LS family parts, and an amazing 23ns when using 74F family parts. Thus, as we have confirmed experimentally with many different computers, simply multiplying the count of port register accesses by the average port register access time yields a good lower-bound estimate of the time taken for each operation.

The port register access counts for a variety of simple operations are given in Table 1. A simple barrier synchronization takes just two port register accesses: one to request synchronization, another to read the synchronization achieved signal. Operations that transmit data as a side-effect of synchronization also can be accomplished with just two port register accesses, but the simplified TTL_PAPERS hardware requires five accesses: one to output the data, two to synchronize and sample the resulting data value, and two more to signal and confirm that all processors have read the result. Because the data path is just four bits wide, larger data transmissions require five port register accesses for every four bits received. Notice that there is no start-up latency; sending K 1-unit data takes the same time as sending 1 K-unit datum.

PAPERS Operation	**4 PEs**	**8 PEs**	*n* PEs
Barrier Synchronization	2	2	2
ANY Test	5	5	5
ALL Test	5	5	5
8-bit Global OR	10	10	10
1-bit Multibroadcast	5	10	5*ceil(*n*/4)
8-bit Broadcast	10	10	10
32-bit Broadcast	40	40	40

Table 1: Port accesses for various aggregate operations

For example, Table 1 predicts that a cluster using four machines, each with 2us port register access time, would take a little more than 4us to perform a barrier synchronization. Experimentally, the difference between actual and predicted lower-bound time is primarily a function of software overhead imposed by slow processors; using a four-machine TTL_PAPERS cluster, the measured times are no more than 5% above the predicted lower bound for 90MHz Pentium, 15% for 33MHz 486DX, 25% for 25MHz 486SX, and 100% for 386DX33 processors. There are also a variety of second-order effects that can hinder actual performance (cache misses, UNIX scheduler anomalies, etc.), but the combined effect has been measured as no worse than about 1.5% per additional processor in the cluster. The overhead from second-order effects can be reduced via various techniques (involving the OS), but are beyond the scope of this paper.

Viewing this data another way, the 90MHz Pentium software overhead for a barrier synchronization was determined to be approximately 160ns. If the processor was directly connected to the PAPERS hardware, without making any other changes, the resulting total operation time could be under 400ns — a full order of magnitude faster than using the conventional parallel port connection. Similarly, the bandwidth could easily exceed 100 Mbits/s, as opposed to the 1.2 Mbits/s maximum of a standard parallel port.

Even using the parallel port interface, the complete system latency, including all hardware and software layers, for TTL_PAPERS is far less than that for most other networks. Typical minimum communication times for local area networks (e.g., Ethernet, ATM, FDDI) using a socket software interface are generally at least a millisecond; even custom parallel machines have relatively high latencies: 32-110 microseconds for the nCUBE2 and 240 microseconds for the Intel Paragon XP/S [1]. Although many of these networks offer significantly higher bandwidth for large block transfers, the combination of low latency and aggregate sampling in a single operation (which would take at least log2(n) communications for other systems) gives TTL_PAPERS better bandwidth for aggregate operations and transmissions of individual data objects. Thus, TTL_PAPERS forms an appropriate complement to such high-bandwidth networks.

4. Conclusion

This paper introduces a new scalable, tightly coupled parallel processing architecture for fine grain execution. PAPERS provides a microsecond latency, low overhead alternative approach for barrier synchronization, global operations, and data communication for MIMD architectures. This architecture can be used to augment high latency, high bandwidth networks and results in a better parallel processing architecture because of the speedup of the global operations.

The demonstration prototypes have shown a 2.5 to 10 microsecond UNIX user process to UNIX user process latency for barrier synchronization. The library of functions shows that a full range of communication operations is possible which allows the TTL_PAPERS unit to execute without additional communication channels.

It has also been shown that removing the many unnecessary layers of software and hardware can result in very low latency. The PAPERS hardware itself has been reduced to the simplest level to increase speed without limiting functionality. In essence, PAPERS has added to MIMD architecture the advantages of SIMD synchronization, global operations, and broadcast communications.

This work was supported in part by ONR Grant No. N0001-91-J-4013 and NSF Grant No. CDA-9015696.

References

[1] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat, "Latency Hiding in Message-Passing Architectures," *8th International Parallel Processing Symposium*, pp. 704-709, 1994.

[2] *Cray T3D System Architecture Overview*, Publication HR-04033, Cray Research, Inc., 2360 Pilot Knob Road, Mendota Heights, MN 55120, 1993.

[3] H. G. Dietz, T. M. Chung, and T. I. Mattox, "A Parallel Processing Support Library Based On Aggregate Communication," *Languages and Compilers for Parallel Computing*, Ed. by C.-H. Huang et al., Springer, New York, pp. 254-268, 1996.

[4] H. F. Jordon, "A Special Purpose Architecture for Finite Element Analysis," *Int'l Conf. on Parallel Processing*, pp. 263-266, 1978.

[5] M. T. O'Keefe and H. G. Dietz, "Hardware barrier synchronization: static barrier MIMD (SBM)," *Int'l Conf. on Parallel Processing*, pp. I 35-42, 1990.

[6] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, "The Implementation of the PASM Prototype Control Hierarchy," *Second Int'l Conf. on Supercomputing*, pp. I 418-427, 1987.

[7] H. Stone, *High-Performance Computer Architecture*, Addison-Wesley, Reading, Massachusetts, 1993.