Bitwise Aggregate Networks

R. Hoare, H. Dietz, T. Mattox, and S. Kim

Purdue University, School of Electrical and Computer Engineering
West Lafayette, IN 47907-1285
[hoare|hankd|tmattox|soohong]@ecn.purdue.edu
http://garage.ecn.purdue.edu/~papers

Abstract

Typical communication networks for parallel processing are based on sending data from one processor to one, or all, of the other processors. Using such a network, many simple operations that require information from every processor requires many point-to-point or broadcast communications. These aggregate operations can be as simple as a barrier synchronization or as complex as an arithmetic reduction. In this paper, we discuss a class of networks that directly implement a wide range of aggregate operations. These networks are capable of performing aggregate operations in a single communication operation using only simple bitwise combining logic in a trivially scalable tree configuration.

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

1. Introduction

For multi-processor systems, shared memory and message passing communication each has a clear heritage. If multiple simple processors are all placed on a single memory bus, communication through shared memory requires no additional logic. If multiple processors do not share a memory bus, the simplest hardware results when one processor sends a message to another through a dedicated link. Both of these designs work very well for small numbers of processors, but become increasingly complex and inefficient as more processors are added to the system. Shared memory systems blossom into complex cache protocols and coherence networks; message-passing systems proliferate routing, arbitration, and switching logic to avoid a multitude of physical point-to-point connections. Thus, both communication methods result in increased latency, achieving reasonable bandwidth only by transmitting a large cache line or message at a time which they admittedly do very well.

We suggest that, at least for the aggregate functions described in this paper, high complexity and latency are not necessary results of using more processors, but rather are side-effects of using a few-processor model for systems that incorporate a large number of processors. Given this, neither the shared memory nor the message-passing model is appropriate.

The question then becomes one of what new model is appropriate. An appropriate model has been in use in parallel software for quite some time. Even a cursory examination of the HPF (High-Performance Fortran) specification quickly reveals a multitude of aggregate functions — functions that collect a set of values (one per processor) and produce either a single result or a set of results. Likewise, both the PVM [6] and MPI [13] libraries define collective operations over a data set provided by a group of processors. Unlike communication using point-to-point mechanisms, the aggregate model discussed in this paper is inherently independent of the number of processors involved.

Using a single clock for all processors makes it easy for many systolic-array and content-addressable memory designs to implement a wide variety of aggregate communication operations using only bitwise logic for communication [5]. This type of implementation of aggregate communication is not only efficient, but is potentially scalable to very large numbers of processors. Custom VLSI implemented SIMD machines often provide at least one global bitwise aggregate communication function. The NCR GAPP [7] has a single bit `GO` (Global Output) that essentially implements a wired NOR across all the processors, and this is used as a basic operation to construct various aggregate functions. The MasPar MP1 and MP2 each implement an eight-bit `globalor` across all the processor elements [11], and use this hardware to implement several aggregate functions: broadcast, any, and bitwise OR reduction.

However, in a machine constructed using processor modules that employ standard cache-based, pipelined, processors in a MIMD configuration, it is very difficult to implement purely synchronous communication by simply sharing a global processor clock. The solution is to use fast hardware barrier synchronization to provide the SIMD-like timing needed. Barrier synchronization can be viewed as an aggregate communication in which each processor outputs a signal, and receives a response signal only after all processors have performed the output operation.

Although most hardware barrier work, from Jordon's original paper [8] to the Cray T3D implementation [1], has not taken good advantage of barriers as a mechanism to support more general aggregate function communication, we suggest that it is both easy and efficient to do so for at least bitwise functions. Further, we note that the logic tree(s) used in implementing efficient barrier synchronization hardware are precisely the same mechanisms that are needed to support bitwise functions. In this paper, we explore some of the bitwise aggregate functions that can be implemented for MIMD processors using only a k bit wide logic tree. Although the logic tree could use nearly any logic function, our examples will focus on NAND logic trees.

NAND is a basic gate for nearly all logic families and is also the mechanism used to construct TTL_PAPERS, the TTL implementation of Purdue's Adapter for Parallel Execution and Rapid Synchronization [4]. TTL_PAPERS is an external unit that links multiple PCs/workstations as a tightly-coupled parallel machine. Performance of the TTL_PAPERS implementation, for each of the bitwise aggregate functions discussed, is also given.

Section 2 discusses the structure and properties of various bitwise aggregate operations. For each operation, the following are given: an explanation of what the operation does, why it is needed, and how it is implemented. Barrier synchronization, broadcast communication operations, reduction operations, voting operations, multi-way branching, and asynchronous parallel signaling are all described. Section 3 describes the TTL_PAPERS hardware that has been built to facilitate these and other aggregate communication operations. Section 4 describes the theoretically minimum hardware that is required for these operations as well as the times achieved for TTL_PAPERS. Section 5 offers conclusions that result from this research.

2. Bitwise Aggregate Operations

A bitwise aggregate operation is a communication that can be implemented by independent logic for each result bit and requires information to be contributed by each processor. The size of this information depends on the requirements of the aggregate operation that is being performed. In many cases, every processor must contribute only a very small amount of information; in some cases, just a single data bit is output by each processor. In contrast, conventional networks generally are designed to optimize transmission of a relatively large data block from one processor to another. Thus, the network structure for bitwise aggregate operations is quite different from that of most networks. This architecture is centered around NAND trees and barrier synchronization.

The remainder of this section discusses a few types of bitwise aggregate operations that all share the property that their implementation can scale to arbitrarily large numbers of processors with negligible change in performance.

2.1. Barrier Synchronization

Barrier synchronization is a common way for MIMD programs to guarantee correct ordering of operations. Specifically, when each processor arrives at a barrier, it waits until all of the other processors have arrived and only then does it continue executing its code. This guarantees that the code after a barrier is only executed after all the processors have finished executing the code that appears before that barrier.

Using traditional point-to-point or broadcast networks, performing a barrier synchronization requires at least N communication operations. Depending on the network topology, some of these communication operations can be executed concurrently so that the time to perform these N communications can be as low as $C * \log2(N)$ where C is the communication time. Using a global bitwise NAND network, a barrier synchronization can be performed in just C time, independent of the number of processors.

The following example demonstrates how a barrier is performed using a multi-bit NAND network. Consider three processors executing a SPMD program in which each processor is to sequentially perform computations A, B, and C with barriers to ensure that no processor can begin executing B until all have completed A, nor C until all have completed B. The execution time for each code block is assumed to be either processor and/or data dependent and thus each barrier is required to guarantee the correct order of execution across the entire machine. As is typical of UNIX-based operating systems, each processor runs an operating system that can pre-emptively interrupt a process to perform context switches or other maintenance tasks.

PE0	PE1	PE2
Execute A	Execute A	Execute A
Out 1	Out 1	Out 1
Poll In for 0	*OS delay*	*OS delay*
Out 0	*OS delay*	*OS delay*
Execute B	Poll In for 0	Poll In for 0
Out 1	*OS delay*	Poll In for 0
Poll In for 0	*OS delay*	Poll In for 0
Out 0	*OS delay*	Out 0
Execute C	Poll In for 0	Execute B

Figure 1: Single NAND tree failed barrier sequence.

2.1.1. Using a Single NAND Tree

Although a single NAND tree is sufficient as a barrier mechanism under the right timing constraints [12], given that processor interactions with the network can be randomly delayed, if only a single NAND tree is used, correct barrier synchronization semantics may not be enforced. For example, in figure 1, all processors will wait (by polling for 0 input) for each of the other processors to signal (by output of 1) that they have completed execution of A. If PE0 is the first processor to see the signal that the barrier has completed, and thus continues to reset its signal (by output of 0) and execute B, PE0 will think that B is being executed by all processors when in fact PE1 and PE2 are both still looking for the signal that PE0 has completed A. This situation only gets worse when PE0 reaches the second barrier, which PE2 then interprets as the first barrier and PE1 again fails to detect. Thus, due to the fact that the processors are each subject to their local OS randomly imposing arbitrary delays (to handle device interrupts and timeshare task switches), each processor could be at a different barrier, yet be unable to discern this fact. In the worst case, a "slow" processor (e.g., PE1) might never move past its first barrier because the other processors always see the barrier completion and reset their output before the slow processor can detect that the barrier has been completed.

2.1.2. Using Two NAND Trees

It can be seen that with random delays, even if two NAND trees, S0 and S1, are used to implement barrier synchronization, proper code execution still cannot be guaranteed. Before a barrier NAND tree is reset, it must be guaranteed that every processor has seen the "barrier done" signal. Therefore, it would seem sufficient to use two barrier NAND trees, such that barrier S0 is reset every time that barrier S1 is reached and barrier S1 is reset every time that barrier S0 is reached. This, however, is not sufficient. Suppose that all of the processors are at barrier S0. After barrier S0 has been completed, the barrier S1 NAND tree will be reset. However, if one processor, PE0, gets ahead of the rest of the processors, it is possible that it will arrive at the next barrier before any of the other processors have been able to reset their signals for barrier S1. Therefore, PE0 will think that barrier S1 has completed and will continue its execution, erroneously assuming that the code before barrier S1 has been completed.

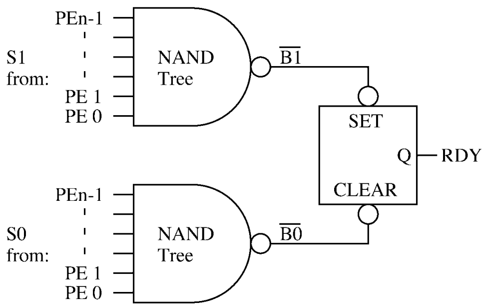

Figure 2: Barrier using two NAND trees and a flip-flop.

2.1.3. Using Two NAND Trees and a Flip-Flop

Barrier synchronization can be performed using three NAND trees, but this requires more signals and logic than using two NAND trees that control a single flip-flop as shown in figure 2. If barrier S0 is able to be reset before barrier S1 is completed and vice versa, then two NAND trees are sufficient to implement barrier synchronization. This can be done by using barrier NAND tree S0 to cause the flip-flop to be reset to a `0` value and barrier NAND tree S1 to cause the flip-flop to be set to a `1` value. In this manner, barrier NAND tree S0 and barrier NAND tree S1 can be assigned in an alternating manner to each barrier that is in the code by first assigning barrier NAND tree S0 to the first barrier, S1 to the second barrier, S0 to the third barrier, etc. For barriers that use barrier NAND tree S0, processors will continue their execution when a `0` is returned and barriers that use barrier NAND tree S1 will continue their execution when a `1` is returned. Therefore, upon reaching a barrier, each processor sends a set-and-reset pair of signals to the barrier NAND trees, S0 and S1. In this way, one barrier is being reset while the other barrier is being set.

For example, let us suppose that all of the processors are between barrier *i* and barrier *i*+1 where barrier *i* is seen to be done by a `0` and barrier *i*+1 is seen to be done by a `1`. When the first processor, PE_A for example, reaches barrier *i*+1, it will send the set-and-reset pair of signals to the barrier NAND trees. The set signal will notify barrier *i*+1 that PE_A has arrived and the reset signal will notify barrier *i* that it should be reset. Now, suppose that there is a processor, PE_B, that has signaled barrier *i* that it has arrived but has not yet seen the "barrier done" signal. PE_A's signal to reset barrier i does not change the status of the flip-flop but does prevent the barrier NAND tree from sending an erroneous "barrier done" signal to the flip-flop and thus does not prevent PE_B from seeing the "barrier done" signal.

2.2. Broadcast

One of the simplest aggregate communication operations is the broadcast of a datum from one processor to all processors.

To accomplish this communication, a single sending processor outputs a datum and all the other processors allow this datum to be passed through the network without changing its value. Using NAND logic, if all but the one broadcasting processor set all their data output bits to 1, the resulting NAND value will be the complement of the value output by the broadcasting processor. Thus, each processor can simply read the NAND result and complement it to obtain the datum broadcast. Alternatively, the sending processor can output the complement of the datum it wants to broadcast, so that the value read by each processor will be precisely the broadcast value. Thus, one NAND bit is needed per datum bit to be transmitted in parallel.

However, this assumes that the data are all sent and received with the proper timing. Unless all processors sample the NAND result at the correct times, a sequence of broadcast data might not be received correctly. This timing constraint can be enforced by using a hardware design which ensures that new data is sampled within a precise time window after detecting a signal transition. Alternatively, and somewhat more robustly, the same effect can be achieved by using an ANDed acknowledge signal closely resembling a barrier synchronization. In fact, timing can be ensured in software simply by using barrier synchronizations to ensure that all processors have read one datum before the sender will broadcast the next.

2.3. Any and All

In data parallel and SIMD-oriented algorithms, it is very common that program control flow is determined by answering questions like "have all processors solved their sub-problems?" or "does any processor need to execute the following operation?" Both `any` and `all` have been widely used for control flow constructs in SIMD languages from Glypnir [10] to MPL [11]. The `all` voting operation enables the processors to detect if every processor has voted "true" for a given decision. Similarly, the `any` voting operation allows a processor to detect if any of the processors has voted "true."

These operations are extremely simple, requiring only a single bit of information from each of the processors. Using point-to-point or broadcast networks requires that a minimum of *N* communications take place, with a minimum execution time of *C**log2(*N*), where *C* is the execution time for a single communication operation. By using a single NAND tree this information can be obtained in a single communication operation that takes only C time. Given that each processor provides a boolean value a which is 1 for "true" and 0 for "false," the basic logic is:

```
Any(a) = OR(a) = NAND(NOT a)
All(a) = AND(a) = NOT (NAND(a))
```

2.4. Reductions (Bitwise, Minimum/Maximum, etc.)

Reductions are aggregate operations in which each processor supplies a value and the result is a single value for the entire collection. Example reductions include minimum value, maximum value, bitwise OR, and bitwise AND. To perform a reduction operation requires at least N communication transactions on traditional point-to-point or broadcast networks with N processors. Depending on the network topology, the time to perform these N communications can only be as low as log2(N) communication times. On several useful reduction operations, significantly lower time complexities can be achieved with a multi-bit NAND network.

Bitwise AND, OR, and NOR reductions can be performed in order one cycles using NAND trees. Specifically, with k NAND trees, k bits of result for an AND, OR, or NOR reduction is produced per cycle. Using NAND trees to operate on a k-bit value a from each processor, and using ˜ to represent bitwise 1's complement, the equivalences are:

```
reduceAND(a) == ~(reduceNAND(a))
reduceOR(a)  ==  (reduceNAND(~a))
reduceNOR(a) == ~(reduceNAND(~a))
```

One very common use of reductions is in optimization problems, where each processor works on a potential solution. Each processor computes a measure of merit or quality for its solution, and then the processors collectively perform a maximum value reduction on these measures. The best solution is the one with the highest score. Genetic algorithms frequently use maximum reductions.

A multi-bit NAND network can perform a maximum or minimum reduction in order one time as well; the time is independent of the number of processors. Specifically, with a k-bit NAND, log2(k) bits of maximum or minimum reduction result are produced per cycle.

In the maximum reduction algorithm, each cycle is a voting operation where each processor votes for a portion of the result bit pattern. To simplify the discussion, let us first consider a maximum reduction over 1-bit unsigned integers. Although one normally thinks of maximum being computed as the result of a series of pairwise comparisons, the maximum of a set of 1-bit unsigned integers is clearly 1 *iff* any processor's value was a 1. As described in section 2.3, an `any` operation is easily accomplished using just a one-bit NAND tree for data transmission. The execution time is independent of the number of processors providing data values.

Now consider a maximum being computed over k-bit unsigned integers. Clearly, determining the most significant bit is exactly the same as performing the 1-bit maximum operation just described, but how can the remaining bits be determined? The answer lies in the fact that all processors know the result of the operation on the most significant bit. If the value for a processor has a most significant bit that does not match this result, then that value cannot be the maximum, and that processor should not participate in determining the values of the less-significant bits. Thus, the k-bit unsigned integer maximum can be accomplished in k `any` operations, with a potentially decreasing set of processors participating in each successive comparison.

This algorithm can be further improved so that the i most significant bits of the maximum can be determined in a single operation. This is done by using m NAND trees for each step, where $m \equiv 2^i$-1. Suppose that the value of a particular processor's most significant i bits is j; then, this processor will output a 0 bit only in the j-1 bit NAND tree. The result is thus a bit vector in which the highest 1 bit indicates the bit pattern that is the maximum for these i most significant bits. Notice that if there is no 1 bit in the result, the most significant i bits must all be zero.

It is perhaps surprising that this same algorithm can be used to compute the maximum (or minimum) over k-bit signed integers or even floating point values. For k-bit signed integers, one can simply add a bias factor of 2^{k-1}, perform the unsigned maximum, then subtract this bias from the result. For floating point numbers, the only additional complication is determining the proper order in which to examine the bits (i.e., exponent before mantissa). For some floating-point formats the comparison can be performed literally as though the value bit patterns were a signed integers.

2.5. Voting To Resolve Resource Contention

Voting operations are aggregate communications in which each processor supplies a "vote," and a cumulative result of the votes is delivered to every processor. There are many different variations on voting; for example:

- Return the processor number of the lowest numbered processor that voted "true". This is done using the minimum reduction algorithm with each processor's value as the processor number (from 0 to N-1) for those that vote "true" and N for those that vote "false."
- Return the approximate number of processors voting "true." The result thus distinguishes whether zero, one, more than one, or all processors voted "true." The approximate count can be obtained by performing an OR reduction of the processor numbers for processors that voted "true" and then collecting three single-bit NAND tree results. One bit computes `any`, thus detecting the case of no "true" voters. Another bit computes `all`, thus detecting the case of all N processors giving "true" votes. The third bit is computed as the `any` over those processors who voted "true" but whose processor number did not match the OR reduction result; this distinguishes between one "true" voter and more than one.
- Return a k-bit vector in which the ith bit indicates how the ith participating processor voted. In general, the ith processor involved in such a vote controls bit i; this is done by output of a 1 bit for all other NAND trees, and output of the

complement of the processor's logical vote value for the *i*th NAND tree.

- Return a *k*-bit vector in which the *i*th bit indicates whether zero or at least one processors want to access the *i*th shared resource.

Despite the differences, all of these operations are typically used to sample global state to obtain information needed to create a static schedule (at runtime) which will be conflict-free.

When several processors need to access a set of shared resources, such as a shared I/O device or software structure (e.g., a data base record), some technique is required to guarantee exclusive access to each processor in turn. By taking a vote before attempting access to a database record, the set of contending processors can be determined. This set of processors can then access the record in a deterministic sequence, with barrier synchronizations between each access to guarantee this ordering. This technique is further described in [2] for statically scheduling access to high-bandwidth conventional networks (e.g., HiPPI).

Voting operations are also quite useful for preventing needless communications. A primary example is a race condition in which several processors attempt to store data into a common data object. Due to the serialization principle, there can be only one final winner of the race condition; thus, if we can detect which processors lose the race, they do not have to send any data. Voting can be used prior to data transmission to determine the winner of the race, and then only the winning processor needs to send data to update the object. A more detailed discussion of this technique can be found in [2].

Various other operations, such as more complex reductions and scans (parallel prefix operations) also incorporate voting steps to enhance performance.

2.6. VLIW-Style Multi-Way Branching

In order to facilitate more aggressive compiler code motions, ideal VLIW (Very Long Instruction Word) machines allow multiple branch tests to be executed in parallel. In the idealized VLIW model, a central control unit collects these test result bits and selects the appropriate branch target. However, as was observed in and [9], it is possible to use bitwise global communication hardware to collect and distribute test results so that each processor can independently perform the corresponding multi-way branch. The necessary communication is actually the second bit vector vote operation described in the previous section.

Figures 3, 4, and 5 show how multi-way jump is implemented with NAND trees. Once the example given in figure 3 has been VLIW scheduled, suppose that PE2 evaluates two branch conditionals and PE0 evaluates one (PE1 and PE3 do not evaluate any). Thus, the voting operation (NAND) has PE0 control the least significant bit and PE2 controls the next two bits. In this way, the low three bits of the NAND result form the densely-encoded key value for every processor to index its local jump table for this multi-way branch. Having the compiler select which processors should control which NAND bits for a given multi-way branch is essentially a trivial register allocation problem. The jump table for PE0 is shown in figure 4, and the corresponding code structure is shown in figure 5.

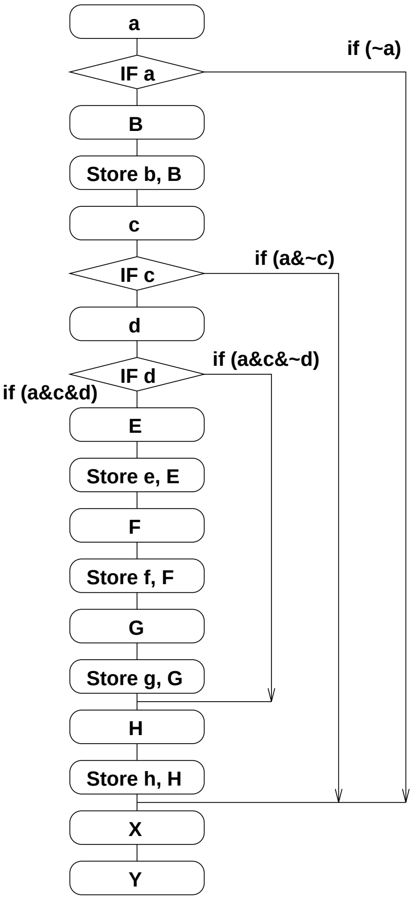

Figure 3: Sequential Code that contains multiple branches.

nand() arguments				*nand() Result*	tab0 [ ]	
PE0	PE1	PE2	PE3		index	label
1111	1111	1111	1111	*0000*	0	L1
1110	1111	1111	1111	*0001*	1	L4
1111	1111	1101	1111	*0010*	2	L3
1110	1111	1101	1111	*0011*	3	L4
1111	1111	1011	1111	*0100*	4	L2
1110	1111	1011	1111	*0101*	5	L4
1111	1111	1001	1111	*0110*	6	L3
1110	1111	1001	1111	*0111*	7	L4

Figure 4: Multi-way branch table.

The resulting code provides not only "branch parallelism" by performing branch tests on PE0 and PE2 simultaneously, but also aggressive code motions by allowing execution of code blocks `B, E, F, G,` and `H` before the four-way branching of figure 3 is resolved.

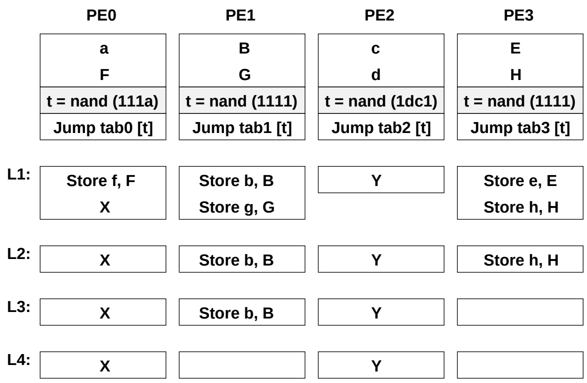

Figure 5: VLIW code based on multi-way branching.

2.7. Asynchronous Parallel Signaling

In order to implement asynchronous communication such as shared memory or message-passing protocols, it is necessary to augment the above fully synchronous operations with a mechanism that can be used to asynchronously trigger a communication. In our discussion of the implementation of barrier synchronization, one can observe that the barrier is composed of at least two operations, the first of which is really a type of asynchronous signal in which each processor signals that it has arrived at the barrier. However, when one thinks of an asynchronous parallel signal one typically thinks in terms of one (or more) processors sending a signal to a group of processors, whereas the first step of a barrier synchronization essentially has all processors signal each other.

The basic parallel signal logic can be implemented using a single NAND tree. To send a signal, a processor simply changes its output bit from 1 to 0, thus causing the NAND tree result to change from 0 to 1. The other processors can then detect this change either by translating this signal transition into a hardware interrupt or by polling to check the state of the result bit. The combining properties of the NAND tree ensure that there is no conflict if multiple processors signal simultaneously. Thus, a processor sending a signal need only output one bit; a processor that is polling need only input one bit.

Again, as discussed for barrier synchronization (section 2.1), there is a potential timing fault if the parallel signal logic could be reset before all processors have responded. For this reason, the TTL_PAPERS hardware incorporates an additional barrier synchronization unit dedicated to parallel signal acknowledge (reset). Alternatively, the signal could be sent using a single NAND tree and other logic could be used to follow it with an ordinary barrier synchronization to serve as an acknowledgment.

3. Hardware Implementation: TTL_PAPERS

TTL_PAPERS [4], is the name for versions of the PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization) hardware which have been carefully engineered to be easily replicated in a university environment. All the TTL_PAPERS versions are built using 74LS TTL logic on a simple one-sided or two-sided circuit board. Despite this simplicity, the TTL_PAPERS 951201 design (figure 6) can be modularly scaled to interconnect thousands of processors.

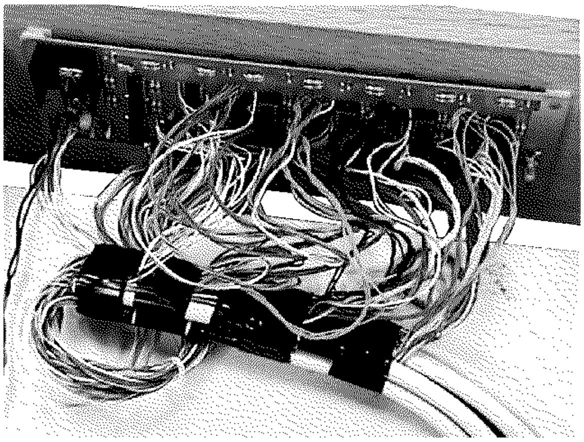

Figure 6: TTL_PAPERS 951201 eight-processor module.

The TTL_PAPERS architecture is comprised of four subsystems: the synchronization subsystem, the parallel signaling subsystem, the data communication subsystem, and the status display subsystem.

3.1. Synchronization Subsystem

TTL_PAPERS uses a variation on the SBM (Static Barrier MIMD) design of [12]. The primary difference between the previously published SBM and the TTL_PAPERS mechanism is that there are two barrier trees and a flip-flop rather than just one barrier tree. The reason is simply that the published SBM silently assumed that the barrier hardware is reset between barriers and, as Section 2.1 describes, this is not the case. In contrast, the use of two trees and a flip-flop (figure 2) allows the hardware for one tree to be reset as a side-effect of the other tree being used.

The obvious implementation of a barrier synchronization would require four I/O cycles: output barrier request, input barrier completed, output barrier acknowledge, and input barrier reset. The TTL_PAPERS barrier unit performs these four functions in two I/O cycles by interleaving and combining these operations for two independent barrier NAND trees. The output barrier request signal for one barrier NAND tree is coupled with the output barrier acknowledge signal for the other. Likewise, both input signals are combined using the flip-flop, resulting in a single input signal that toggles as described in section 2.1.3.

Because this design needs no clocking, packetization, arbitration logic, etc., but merely needs a couple of NAND trees and a flip-flop, the design trivially scales to very large numbers of processors without introducing significant complexity or delays.

3.2. Communication Subsystem

The TTL_PAPERS library provides a rich array of aggregate communication operations. Due to the synchronous nature of the data communications, the communication subsystem can rigorously schedule which processor is sending data and in which order. Further, all communications are inherently reliable; no data is ever lost nor is there ever a need for error correction or retransmission. This allows the communication architecture to be extraordinarily simple. Thus, buffers, collision detection logic, arbitration logic, and interrupt handlers are not needed.

TTL_PAPERS uses a combination of hardware and software to provide a wide range of aggregate communication operations. The functions described in this paper are all directly implemented. More complex operations, such as double-precision floating point multiply scans (parallel prefix), are implemented using software that takes advantage of the directly implemented functions both to transmit data and to obtain global state information that allows more effective algorithms to be applied. Of course, constructed communication operations are slower than those directly implemented, but most aggregate functions on small (64-bit or smaller) data objects still execute in less time than it takes for a single minimum communication using traditional networks [3].

Like the barrier structure, this hardware design needs no clocking, packetization, arbitration logic, etc., but merely a group of NAND trees. Thus, the design trivially scales to very large numbers of processors without introducing significant complexity or delays.

3.3. Parallel Signaling Subsystem

As suggested in section 2.7, the parallel signaling subsystem in TTL_PAPERS is very similar to the barrier subsystem. An "interrupt request" NAND tree is connected to a flip-flop so that any processor can cause the flip-flop to register a pending signal. A second NAND tree, which acts to provide a barrier synchronization, allows all processors together to acknowledge and reset any pending signal(s). Due to parallel port limitations, each processor has an additional output bit which is used to select whether the normal, toggling, barrier synchronization input is visible or the signal input is visible on its input port.

Because it is virtually identical to the barrier subsystem, the parallel signaling unit is also very scalable.

4. Performance

All the operations discussed in this paper are bitwise aggregates that can be computed using a fixed number of NAND trees, independent of the number of processors being interconnected. If we assume that these NAND trees are implemented using TTL-compatible logic in a single, centralized, hub, the propagation delay is barely affected by the size of each NAND tree; even thousands of processors could be accommodated with a logic delay on the order of 100 nanoseconds. Depending on physical layout, the delays introduced by the wires or cables between each processor and this hub may reach several hundred nanoseconds, but are essentially independent of the total number of processors. The device register access time is relatively large using standard processors, typically between 300 nanoseconds and 2 microseconds depending on bus interface (e.g., PCI and ISA bus bridges), but it is also independent of processor count.

	Minimum NAND Trees	Minimum I/O Cycles
Barrier Sync.	1	2
k-bit Broadcast	t+1	2*ceil(k/t)
Any	2	2
All	2	2
k-bit OR	t+1	2*ceil(k/t)
k-bit AND	t+1	2*ceil(k/t)
k-bit NAND	t+1	2*ceil(k/t)
k-bit NOR	t+1	2*ceil(k/t)
k-bit Maximum	t+1	2*ceil(k/floor(log2(t+1)))
k-bit Minimum	t+1	2*ceil(k/floor(log2(t+1)))
k-bit vector Vote	t+1	2*ceil(k/t)
Async. Signal	1	1

Table 1: Hardware and time complexity using NAND trees.

Given this, there is little to discuss about how complexity and performance scale with system size. Each NAND tree has the obvious structure, an N-input NAND tree whose output is duplicated by a fan-out tree of drivers. Because the only direct impact of scaling on network speed is a small variation in NAND tree delay, which can be a negligibly small fraction of the total delay, it is reasonable to consider the performance per processor to be essentially independent of machine size. Tthis fact has been confirmed empirically in our experiments.

However, different aggregate operations can show different performance using the same tree logic, cable, and interface. One cause of these differences is the number of NAND trees available for the operation. Some aggregate functions need, or can improve performance by using, multiple NAND trees. The minimum number of NAND trees for each bitwise aggregate operation is given in table 1. In this table, t is used to represent the number of NAND trees being used for data transmission, which must be at least 1. The other primary characteristic is the number of I/O device register accesses (bus I/O cycles) required for each aggregate function. These also are listed in table 1, parameterized by t and by k, the number of data bits from each processor.

The performance figures listed in table 1 are based on the assumption that each processor is interfaced to the NAND trees in a way that ensures that each processor will be able to immediately react to any state change on its inputs. This is a reasonable model if the processor is not subject to interrupts (caused by other devices or timesharing) or is connected using "smart" interface hardware that can immediately react to input state

	TTL_PAPERS I/O Cycles	Measured Time for k=32 (μs)
Barrier Sync.	2	2.5
k-bit Broadcast	5*ceil(k/4)	59
Any	5	6.3
All	5	6.3
k-bit OR	5*ceil(k/4)	59
k-bit AND	5*ceil(k/4)	59
k-bit NAND	5*ceil(k/4)	59
k-bit NOR	5*ceil(k/4)	59
k-bit Maximum	5*ceil(k/2)	112
k-bit Minimum	5*ceil(k/2)	112
k-bit vector Vote	5*ceil(k/4)	59
Async. Signal	1	1.2

Table 2: Predicted and measured time using TTL_PAPERS.

changes.

However, using standard parallel port (SPP) interfaces to connect conventional PCs or workstations each running UNIX, additional overhead is introduced. For TTL_PAPERS, this overhead is summarized in table 2. TTL_PAPERS maintains two-cycle barrier synchronization performance by using two NAND trees and a flip-flop. Although the SPP provides up to 12 output bits, it only provides five bits of input. TTL_PAPERS uses one of the five bits to detect each barrier synchronization, leaving four bits for data transmission. For reliable data transmission using the SPP connection, a five-cycle communication is needed. This sequence uses two two-cycle barrier synchronizations to ensure that all data is available before the result is read and is not changed until all processors have read the result. The fifth operation simply prevents a race between data bits and barrier request bits.

The measured execution times quoted in table 2 were obtained using a TTL_PAPERS cluster of four 486DX33 PCs, each running Linux. In comparison, using Ethernet and PVM 3, the same cluster yields a barrier synchronization time of 49,000 μs and aggregate function times around 100,000 μs. Thus, the speed difference ranges from three to four orders of magnitude in favor of TTL_PAPERS, and this performance gap widens for larger clusters.

5. Conclusion

In this paper, we have suggested that the type of network once recommended for use in bit-serial systolic array and custom VLSI SIMD architectures can be readily and effectively applied to interconnect MIMD processors — or even clusters of PCs or workstations. Although such networks were traditionally viewed as requiring a purely synchronous machine organization, they trivially implement efficient barrier synchronization, thus providing this synchronous environment without demanding any special processor features.

Despite the strikingly simple hardware, the performance (discussed in section 4) on completely scalable aggregate function communications is dramatically superior to that of far more complex conventional networks. Not listed in this paper are the multitude of additional aggregate functions for which scaling to large numbers of processors causes some slowdown, but performance is again far better than that of conventional networks that optimize bandwidth for large-block transmissions.

References

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

[2] H. G. Dietz, W. E. Cohen, T. Muhammad, and T. I. Mattox, "Compiler Techniques For Fine-Grain Execution On Workstation Clusters Using PAPERS," 7th Annual Workshop on Languages and Compilers for Parallel Computing (also to appear as a book chapter from Springer Verlag), Cornell University, August 1994.

[3] H. G. Dietz, T. M. Chung, and T. I. Mattox, "A Parallel Processing Support Library Based On Synchronized Aggregate Communication," *1995 Workshop on Languages and Compilers for Parallel Computing*, Ohio State University, Ohio, August 1995.

[4] H. G. Dietz, R. Hoare, and T. Mattox, "A Fine-Grain Parallel Architecture Based On Barrier Synchronization," *International Conference on Parallel Processing*, August 1996.

[5] C. C. Foster, *Content Addressable Parallel Processors*, Litton Educational Publishing, Inc., New York, 1976.

[6] A. Geist, et. al., *PVM, Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* , The MIT Press, Cambridge, Massachusetts, 1995.

[7] W. Hannaway, G. Shea, W. R. Bishop, "Handling real-time images comes naturally to systolic array chip," *Electronic Design*, November 15, 1984.

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

[9] S. P. Kim and H. G. Dietz, "VLIW-Style Parallelism On A Workstation Cluster, " *submitted to International Conference on Parallel Architectures and Compilation Techniques*, Boston, MA, 1996

[10] D. H. Lawrie, I. Layman, D. Baer, and J. M. Randal, "The Glypnir Language," Communications of the ACM, Vol. 18, No. 3, March 1975.

[11] MasPar Computer Corporation, *MasPar Programming Language (ANSI C compatible MPL) Reference Manual, Software Version 2.2*, Document Number 9302-0001, Sunnyvale, California, November 1991.

[12] M. T. O'Keefe and H. G. Dietz, "Hardware barrier synchronization: static barrier MIMD (SBM)," Proc. of 1990 Int'l Conf. on Parallel Processing, St. Charles, IL, pp. I 35-42, August 1990.

[13] M. Snir, et. al., *MPI The Complete Reference* , pp. 147-199, The MIT Press, Cambridge, Massachusetts, 1996.