A Parallel Processing Support Library Based On Synchronized Aggregate Communication

H. G. Dietz, T. M. Chung, and T. I. Mattox

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285
hankd@ecn.purdue.edu
April 1995

Abstract

The concept of "data parallelism" is a pervasive force throughout parallel processing. Although a certain level of processing-element autonomy can help performance, the fact is that many parallel algorithms, applications, and compiler analysis techniques focus on identifying a set of data objects that can be processed using loosely synchronous parallelism. Thus, it is not surprising that a large number of communication libraries support at least a few synchronized aggregate operations on data. In this paper, we present an overview of the parallel processing support library for PAPERS clusters.

Unlike most other systems, which construct aggregate communications by layering message-passing or shared-memory communication, the PAPERS hardware and software directly implements a model of parallel execution based on synchronized aggregate communications. Asynchronous processor operation is fully supported, but asynchronous communications are not directly supported (although they can be derived using the PAPERS parallel interrupt mechanism). Thus, PAPERS provides highly efficient aggregate communications for MIMD, SIMD, and VLIW execution modes. We demonstrate the effectiveness of this implementation by presenting detailed benchmarks for a variety of other libraries supporting dedicated parallel machines and workstation clusters.

This work was supported in part by the Office of Naval Research (ONR) under grant number N00014-91-J-4013 and by the National Science Foundation (NSF) under Parallel Infrastructure Grant number CDA-9015696.

1. Introduction

PAPERS, Purdue's Adapter for Parallel Execution and Rapid Synchronization, is inexpensive hardware designed to provide very low latency barrier synchronization and aggregate communication operations to user-level unix processes running on a cluster of personal computers or workstations. The PAPERS library is designed to provide both:

- Good support for hand-coding of parallel applications and
- An efficient target for optimizing/parallelizing compilers.

Of course, these goals can only be accomplished by avoiding the usual software interface layering and the associated increases in latency.

Thus, the PAPERS library must solve two fundamental problems. The first problem is how to avoid the usual ill effects of using a cluster, as we discuss in Section 1.1. The second, discussed in Section 1.2, is how to construct a parallel execution model that can provide the appropriate functionality without sacrificing performance.

1.1. Clustering

The use of clusters of workstations/PCs for parallel computation is an attractive idea for many reasons. However, there are also many reasons why such clusters do not make effective parallel computing platforms. The most obvious problem is that of efficiently transmitting data between the processing elements (PEs) of the cluster. Traditional networks, ranging from Ethernet to FDDI and ATM, were all designed to maximize bandwidth rather than to minimize latency. This is not just a matter of hardware latency, but also of software latency. Standard network communication protocols (e.g., BSD sockets) require at least one system call and process context switch for each operation. This often adds as much as 100 microseconds to the latency of each operation. Libraries like PVM, which introduce even more software layers, can add milliseconds of delay to each operation. Given such high latency, only very coarse-grain parallelism is effective.

In contrast, everything about PAPERS is designed with the goal of supporting fine-grain parallelism. Although a PAPERS cluster might use the exact same personal computers and workstations that a conventional cluster would employ, there is only one software layer between the PAPERS hardware and the user code: the PAPERS library. This library executes as part of the user process, directly accessing the PAPERS hardware either by executing port I/O instructions or by volatile references to memory-mapped device registers. There is no system call required for a typical operation. The PAPERS hardware provides such low latency that, if we had used a system call for each PAPERS operation, the average latency would have been between 5 and 50 times longer. In fact, the simplest PAPERS operations are so fast that they are implemented by in-line code to avoid the overhead of a subroutine call.

Another interesting implication of the PAPERS mechanism and software interface is that the execution time of each operation is very consistent and predictable. This property makes it possible for compiler analysis to more precisely estimate the performance of alternative codings, making static scheduling far more effective than it would be for a traditional cluster. In fact, PAPERS provides such a precise time reference that it can even be used to support scheduling of hard real-time operations; latency may be a few microseconds, but timing accuracy (repeatability or clock synchronization) is typically within a single microsecond.

1.2. Parallel Computation Model

The PAPERS library is not compliant with emerging standards for message-passing libraries (e.g., PVM [GeS93] or MPI [Mes94]) because the basic operations supported by the PAPERS hardware are **not** based on a message-passing model. While interactions between PEs using message-passing systems are often asynchronous and are always initiated by individual PEs, PAPERS operations are aggregate operations which are requested and performed in unison by all PEs within a specified group. Thus, although a PAPERS cluster is not restricted to SIMD execution, all PAPERS operations are based on a model of aggregate interaction that closely resembles SIMD communication. This is not a coincidence; years of research involving Purdue University's PASM (PArtitionable Simd Mimd) prototype supercomputer have experimentally demonstrated that SIMD-like interactions are more efficient than traditional message-passing for a wide range of applications [BeS91].

The PAPERS execution model is based on the properties of the generalized barrier synchronization mechanism that we have been developing since [DiS88]. In barrier MIMD execution, each processor executes independently until it wishes to synchronize with an *arbitrary group* of (one or more) other processors. The barrier synchronization is accomplished by each processor individually informing the barrier unit that it has arrived, and then waiting to be notified that all participating processors have arrived. Once the barrier synchronization has completed, execution of each processor is again completely independent. Thus, barrier MIMD execution is a purely MIMD execution model that can, at any point desired, impose timing properties accurately emulating either a SIMD or VLIW execution model.

Suppose that, instead of executing just a barrier wait, all participating processors may additionally perform a communication operation: communication is literally a side-effect of an arbitrary set of PEs executing a barrier synchronization. This communication model is somewhat like synchronous message-passing in concept, but is much more powerful because communication need not be point-to-point. In this model, the PAPERS unit collects an aggregate of the data items output by each PE that participates in the current barrier. The value obtained by each PE can thus be any of a wide range of functions of the aggregated data — and these functions can be *computed in parallel within the PAPERS unit*. For example, reductions, scans (parallel prefix operations), and various other functions can yield performance comparable to or better than the cost of performing a simple permutation communication using a more traditional network. Current PAPERS hardware directly implements only a subset of all possible functions, but the high performance achieved with remarkably simple hardware clearly demonstrates the power of the model.

Despite the synchronous nature of aggregate communication, it is important to remember that only the specified group of PEs must participate in each communication, and every PE can be executing a unique program. In fact, each PE participating in a particular aggregate communication even can behave differently within that communication (e.g., one processor broadcasts while others receive). However, unlike asynchronous message-passing, the role of each processor in a synchronization or communication operation must be determined by that processor. Processors are free to dynamically alter how they interact, but only by means that preserve the static qualities of the interactions. For example, the only way for processor A to get data from processor B is for B to know that some processor might want this data, and to act accordingly in aggregate with A.

The PAPERS hardware also provides a parallel interrupt mechanism that can be used by any processor to signal the selected group of processors, and this device could be used to simulate asynchronous message-passing. However, the performance characteristics of asynchronous message-passing using PAPERS in this way are similar to those obtained using PVM. Thus, we do not include any such messaging operations in this paper, nor do we consider such operations to be properly part of the PAPERS library communication model. PAPERS parallel interrupts are normally used only by PEN, the Papers ENvironment that performs high-level parallel OS functions such as gang scheduling of parallel programs.

In summary, the PAPERS library model is neither message-passing nor shared memory, but is an inherently different model based on the concept of synchronously aggregating data from an arbitrary group of PEs. We suggest that this barrier MIMD based model yields simple and efficient hardware that better supports the needs of many parallel applications executing in any combination of MIMD, SIMD, and VLIW execution modes.

2. Overview

The PAPERS library discussed in this document is designed to provide all the basic components needed to support a wide range of parallel applications. The routines are intended to be called directly within C code to be compiled by the Gnu C Compiler. Thus, this library can be used:

- Directly by the application programmer using C or any language that can be translated into C (e.g., Fortran via `f2c`, Pascal via `p2c`).
- By an optimizing/parallelizing compiler generating stylized C code as a "portable assembly language" target. We have experimentally confirmed that expressing tuples as C code written using temporary variables declared with the `register` attribute results in an efficient, and predictable, translation to native assembly code. For example, the PAPERS MPL compiler generates code in this style. This approach allows instruction-level analysis and optimization to be easily parameterized to account for differences between machines — which is particularly important if the machines within a cluster are heterogeneous.

In this paper, the basic PAPERS library functions are described and benchmarked in detail. The PAPERS library also contains other operations such as support for parallel file I/O; however, these routines are beyond the scope of the current work.

Before describing the library routines themselves and their performance relative to comparable routines in other libraries, it is useful to review the basic data types managed by the routines. This is done in Section 2.1. Likewise, the performance evaluation requires some background discussion of the methods used to measure performance and scalability. Section 2.2 reviews these issues.

2.1. Data Types

Unlike most implementations of PVM [GeS93] or MPI [Mes94], the PAPERS hardware and interface software does not have a significant set-up time; an operation on *n* units of data generally takes nearly the same time as performing *n* single-unit data operations. This fact is clearly demonstrated by the benchmark data given in the following sections. Thus, there is no reason to pack/unpack blocks of data into a memory-based message buffer. It is actually more efficient to perform the PAPERS operations directly using processor registers, with one PAPERS operation performed for each datum being acted upon, because this saves the cost of having to read and write each datum to a local memory buffer. In fact, the time to transmit a datum from a processor register through a PAPERS unit could be comparable to the time taken for two local memory buffer accesses.

Thus, unlike PVM or MPI, the PAPERS library routines focus on low-latency communication operations that can act upon individual data values within a computation. Functions that return data values have multiple versions, one for each basic data type supported by the Gnu C Compiler (i.e., each type that could be held in a processor register). Each of these types is listed in Table 1, along with the equivalent portable type definition used in the PAPERS library and the corresponding function name suffix. For example, the function that sums a 32-bit `unsigned int` value from each participating processor is called `p_reduceAdd32u()`, where the suffix `32u` identifies the type of both the operand and the return value.

GCC Type	PAPERS Type	PAPERS Suffix
boolean (`unsigned int`)	`uint1`	`1u`
8-bit `char`	`int8`	`8`
8-bit `unsigned char`	`uint8`	`8u`
16-bit `short`	`int16`	`16`
16-bit `unsigned short`	`uint16`	`16u`
32-bit `int`	`int32`	`32`
32-bit `unsigned int`	`uint32`	`32u`
64-bit `long long`	`int64`	`64`
64-bit `unsigned long long`	`uint64`	`64u`
32-bit `float`	`f32`	`f`
64-bit `double`	`f64`	`d`

Table 1: PAPERS Data Types

The portable definition of these types yields a minor complication: if the machines within a cluster are not all based on the same processor, each may use a slightly different representation for the same value. In PVM 2, for example, this was handled using the notoriously slow UNIX socket XDR (eXternal Data Representation) library routines to convert to a standard exchange format as values are moved into or out of the message buffer in local memory. However, the PAPERS unit is fed data in a way that is inherently insensitive to memory byte order; each datum is apparently sent directly from a processor register to a processor register. Thus, all integer data are inherently portable, and byte-order variations on floating point formats are also handled correctly. Currently, we assume that the floating point formats used on different machines do not differ by more than byte ordering — an assumption that holds true for the vast majority of workstations. Later versions of the PAPERS library may handle more drastic floating point format variations by sending floating point values as separate exponent and mantissa, because these can be portably obtained and sent faster using PAPERS than the standard XDR routine could be applied. Of course, none of these problems arise in homogeneous clusters or dedicated parallel machines.

A similar portability constraint arises with respect to taking advantage of the fact that PAPERS may be able to operate faster on lower-precision values by transmitting only the specified number of bits. Although it might be possible to, for example, send only a portion of the mantissa of a floating point value, such operations are not portable. Thus, only functions that operate on unsigned integer values can be augmented by versions that have the ability to restrict the PAPERS operations to a specific number of bits. These functions are named with the suffix `Bits` followed by the suffix that specifies the unsigned integer type from which the low bits are extracted. The benchmark data presented in this paper ignores the `Bits` operations because none of the other libraries supports any such operations.

We have also omitted benchmarks of the `1u` routines, which are used to manage 1-bit logical true/false values. Most of these routines take the same time as `p_any()`. Further, none of the other libraries directly supports such operations.

2.2. Performance Analysis

Although it is very difficult to argue against the basic concept of reducing hardware and software overhead, the primary purpose of the performance analysis in this paper is to show how effective the PAPERS library is in comparison to other libraries and other hardware. As a secondary issue, because these operations are portable at least across the listed platforms, these performance numbers also can provide a useful guide for estimating the performance of codes being ported from one platform to another.

We were able to obtain at least some performance numbers for four different libraries running on dedicated commercial supercomputers. The libraries benchmarked were:

- Cray T3D [Cra93]. A 4 PE partition was benchmarked. This library supports very few of the aggregate operations, but is very effective in implementing these few operations.
- Cray T3D (PVM). A 4 PE partition was benchmarked. At the time we conducted these benchmarks, this PVM-compatible library supported only those aggregate operations supported in the standard Cray library.
- MasPar MP-1 MPL support library. The MasPar that we benchmarked is a 16,384 PE SIMD machine that was run with only 4 PEs enabled. It is significant, however, that most operations become only slightly slower when more PEs are enabled.

- Paragon XP/S [Int94]. The Paragon that we benchmarked is a 140 PE MIMD machine, but the benchmarks reported were run on a 4 PE partition. We found that execution time of most operations increased nearly linearly as the number of PEs in the partition is increased.

For our library running on a PAPERS cluster, we actually benchmarked two different versions of the PAPERS hardware using the same four personal computers as PEs. Each PE was an IBM ValuePoint 486DX33 system running Linux 1.1.75. The PAPERS units were:

- PAPERS1. A PAL-based version of PAPERS that connects to 4 PEs via their parallel ports. This unit supports the full dynamic barrier mechanism [OKD90a] (implemented as described in [Muh95]) and provides hardware support for both `putget` and NAND aggregate operations. Because the external data path for each PE is just 4 bits wide, only operations that require each PE to obtain more data as the design is scaled to larger systems incur proportionally more delay.
- TTL_PAPERS [DiC95]. A TTL-based version of PAPERS that connects to 4 PEs via their parallel ports. This design has been greatly simplified to make it cheaper and easier to build, so performance of TTL_PAPERS generally marks the lower bound on PAPERS performance. This unit supports the full static barrier mechanism [OKD90] and provides hardware support for NAND aggregate operations. Because both the internal and external data paths are just 4 bits wide, operations that require more data to pass through the unit as the design is scaled to larger systems incur proportionally more delay.

To provide a reference for the performance of a more traditional cluster parallelism library, we also benchmarked performance using the same PEs that we used to benchmark the PAPERS units listed above. These four IBM ValuePoint 486DX33 systems running Linux 1.1.75 were connected using a standard Ethernet (with insignificant other traffic on the Ethernet). The library benchmarked was:

- PVM 3.3.7 [GeS93]. This library is the de-facto standard for parallel computing on workstation clusters, and this version generally yields significantly better benchmark times than older versions of PVM. Because communication is using Ethernet, the execution time of most operations increases nearly linearly as the number of PEs is increased.

Because appropriately detailed performance measurements are very rarely collected and published, we found it necessary to conduct nearly all of the benchmarking ourselves. To minimize any bias, we have used the standard library routine without modification for each supported operation, and indicate times obtained for operations that were not directly supported by following the time with the "`*`" character; a "`?`" character is given instead of a time in cases where the routine was not directly supported and we were unable to create an efficient implementation. We also found that some operations could not be supported without major restructuring, in which case we list a "−" character rather than an execution time.

All the tables in the remainder of this document have the same form. Each line corresponds to the system and library being measured. Each column corresponds to the operation being performed. Operations within the same table that differ only by the size and type of their operand/return value are indicated by giving the size of the integer value in bits or listing "Float for 32-bit floating-point and "Double" for 64-bit floating-point. (The timing difference between signed and unsigned integer operations is generally negligible.) **All execution times are listed in microseconds.**

3. PAPERS Management Operations

`p_init()`	Check-in with PAPERS hardware
`p_exit(e)`	Check-out with PAPERS hardware, exit code e
`p_error(s)`	Print error message s, check-out, and exit

Before any program can use the PAPERS hardware, it must:

- Obtain PAPERS access permission from the operating system.
- Confirm that the software and hardware environment is consistent.
- Generate an initial barrier synchronization.

All of these functions are performed by simply calling `p_init()`.

To indicate that a program has completed using PAPERS, it calls `p_exit()` with an argument which is used much like the argument to the ordinary UNIX `exit()` call. Notice, however, that calling `p_exit()` does not necessarily terminate the calling process — it just terminates the use of PAPERS. The routine `p_error()` is provided to terminate the current PAPERS parallel program with the given error message.

4. Low-Level PAPERS Operations

`p_enqueue(m)`	Enqueue the barrier mask m
`p_wait()`	Barrier synchronize with current mask
`p_waitvec(f)`	Return bit vector assembled from f of each active processor
`p_any(f)`	Did any enabled processor have f true?
`p_all(f)`	Did all enabled processors have f true?

At the lowest level of the PAPERS library, these routines form the interface for the most direct use of the PAPERS hardware.

To set the current barrier group, `p_enqueue()` is called with a bit mask that has a 1 bit in each position corresponding to a member of the barrier group. Currently, by default, all available processors are in the initial barrier group. To perform a barrier wait with the current barrier group, simply call `p_wait()`.

The remaining three functions all operate on integer flags as single-bit truth values. The `p_any()` and `p_all()` operations are familiar enough to any user of parallel processing.

The `p_waitvec()` operation is actually a one-bit multibroadcast that allows all processors to obtain a mask vector constructed from the one-bit votes of all processors. Processor *k*'s vote is in bit *k* of the result. Enabled processors vote 1 if f is true (non-zero), 0 otherwise. Disabled processors always vote 0. This voting operation was originally devised so that processors could vote on which partition they wanted to be in when a barrier group is divided by a parallel conditional, trivially creating the new barrier mask for each processor. However, it is also used to determine things like which processors want to access a shared resource, or which have extra work to do, etc. Thus, `p_waitvec()` is very commonly used, for example, by code generated by compilers or translators to schedule I/O operations; however, users are not expected to directly access this type of function.

The times measured for these functions are listed in Table 2. Note that the Cray T3D hardware barrier mechanism does not support arbitrary partitioning, but only static partitioning on fixed boundaries; thus, the concept of dynamically enqueuing a new barrier grouping does not exist. Likewise, both the Paragon XP/S and PVM 3 treat groupings as essentially static. However, as a SIMD machine with enable masking hardware, the MasPar MP-1 supports runtime regrouping of PEs very efficiently. The MasPar's completely synchronous nature explains the remarkable Wait time, and its global OR network makes the other times also very respectable.

	Enqueue	Wait	Waitvec	Any	All
Cray T3D	-	1.7	?	?	?
Cray T3D (PVM)	-	21	?	?	?
MasPar MP-1	4.7	0.1	9.4	2.7	5.1
Paragon XP/S	-	530	?	?	?
PAPERS1	3.5	3.1	3.2	3.2	3.3
TTL_PAPERS	0.1	2.5	6.3	6.3	6.3
PVM 3	-	49,000	?	?	?

Table 2: Timing for Low-Level Operations (microseconds)

5. Putget Operations

`p_putget`*type*`(d, s)` Put d, get (return) d from PE s

The PAPERS library putget operation is the most fundamental mechanism for supporting arbitrary communication between processors. The model is simply that each processor in the current barrier group outputs a datum, specifies which processor it wants the datum from, and returns the selected value. Logically, each processor writes to its own bus buffer within the PAPERS unit and simply snoops the value written on the bus it selects.

Although some versions of PAPERS directly implement this mechanism (e.g., PAPERS1), TTL_PAPERS does not. Instead, it must map the putget into a series of simple broadcasts (see Section 9). There are two basic approaches to this mapping. One method is to simply force every processor in the barrier group to perform a broadcast; this is simple and consistent, but is inefficient for patterns in which the values sent by some processors are not used (i.e., this technique is best for permutations). The alternative is to first collect a vote from the processors as to which PEs would be sending useful values, and then to send only those values. Surprisingly, this voting operation is trivially accomplished for *n* processors using *n*/4 4-bit NAND operations to accumulate a vote bit vector in which a 1 bit corresponds to a processor that at least one PE wants the value from, and a 0 bit indicates that the corresponding PE need not send any data. In fact, this voting operation is just `p_waitvec()`.

The timing of putget operations is particularly important because this operation is used to implement arbitrary communication patterns. For example, a single putget can implement any permutation or any multi-cast. There is no concept of "neighbors" or network contention for putget — every processor is just one communication away.

Table 3 gives the times for putget operations. Clearly, the MasPar's router network supports the notion of putget surprisingly well. What is not evident in this table is that even the times for the Paragon and PVM are essentially independent of the communication pattern implied by the putget.

	8	16	32	64	Float	Double
MasPar MP-1	33	36	44	67	44	67
Paragon XP/S	710*	700*	700*	700*	700*	700*
PAPERS1	8.5	15	27	62	29	63
TTL_PAPERS	58	111	216	434	219	467
PVM 3	97,000*	96,000*	100,000*	100,000*	96,000*	101,000*

Table 3: Timing for Putget Operations (microseconds)

6. Gather Operations

`p_gather`*type*`(p, d)` Gather an array p[*i*] = d from PE *i*

The PAPERS library gather operations collect a datum from each processor in the current barrier group, and fill-in the corresponding entries in each processor's local memory array whose address is passed as the first argument to the function. Array entries corresponding to processors that are not in the current barrier group are unaffected by the gather. Thus, a gather operation is essentially implemented by performing either *n* broadcast operations (as in TTL_PAPERS) or *n*-1 putget operations (as in PAPERS1).

There is no scheduling overhead introduced for TTL_PAPERS to schedule the broadcasts because each processor in the group has a local copy of the barrier bit mask. Thus, no communication is needed to agree on which processor will broadcast when. The order is from the lowest-numbered enabled PE to the highest, determined independently by each processor shifting through its local copy of the barrier bit mask. Likewise, PAPERS1 putgets are scheduled without additional overhead.

The performance of these functions is summarized in Table 4.

	8	16	32	64	Float	Double
MasPar MP-1	125*	144*	191*	290*	191*	290*
Paragon XP/S	710*	700*	700*	700*	700*	700*
PAPERS1	33	53	89	194	95	199
TTL_PAPERS	57	110	216	439	248	444
PVM 3	96,000*	96,000*	98,000*	99,000*	98,000*	100,000*

Table 4: Timing for Gather Operations (microseconds)

7. Arithmetic Reduction Operations

`p_reduceAdd`*type*`(d)`	Return the sum of d from each PE
`p_reduceMul`*type*`(d)`	Return the product of d from each PE
`p_reduceMin`*type*`(d)`	Return the minimum of d from each PE
`p_reduceMax`*type*`(d)`	Return the maximum of d from each PE

By definition, a reduction operation is an associative operation; i.e., the order in which the component operations are performed does not significantly alter the resulting value. One way to perform these operations is to have all processors collect all the data and then serially reduce the local copy. This yields a very simple implementation suitable for clusters with up to about 8 processors, and this technique is currently used for the 4 PE TTL_PAPERS library. For example, all the basic variations on `p_reduceAdd` can be implemented by:

```
#define REDUCEADD(name, type, suffix) \
type \
name(register type datum) \
{ \
  type buf[NPROC]; \
  buf[0] = (buf[1] = (buf[2] = (buf[3] = 0))); \
  p_gather##suffix(&(buf[0]), datum); \
  return(buf[0] + buf[1] + buf[2] + buf[3]); \
}
```

```
REDUCEADD(p_reduceAdd8, int8, 8)
REDUCEADD(p_reduceAdd8u, uint8, 8u)
REDUCEADD(p_reduceAdd16,int16, 16)
REDUCEADD(p_reduceAdd16u, uint16, 16u)
REDUCEADD(p_reduceAdd32,int32, 32)
REDUCEADD(p_reduceAdd32u, uint32, 32u)
REDUCEADD(p_reduceAdd64,int64, 64)
REDUCEADD(p_reduceAdd64u, uint64, 64u)
REDUCEADD(p_reduceAddf, f32, f)
REDUCEADD(p_reduceAddd, f64, d)
```

The problem with using this code on larger clusters is that it executes in $O(n)$, rather than $O(\log_2 n)$, time for n processors. This is remedied by using the usual tree reduction with communication implemented by a putget operation for each of the $\log_2 n$ communication steps. However, the tree reduction is somewhat complicated by the fact that some processors might not be members of the current barrier group, in which case they do not participate in the reduction. Thus, if a tree reduction is to be used, the participating nodes must effectively be "renumbered" to determine their positions within the reduction tree. Fortunately, the renumbering can be computed without any communication operations — each node is numbered by `p_enumerate()` and the tree contains `p_population()` leaves (see Section 11). This is the method used by PAPERS1.

Benchmark times for sum reduction, product reduction, minimum reduction, and maximum reduction are presented in Tables 5 through 8.

	8	16	32	64	Float	Double
MasPar MP-1	124	139	180	266	302	428
Paragon XP/S	700*	700*	700	700	710	680
PAPERS1	20	33	57	128	63	131
TTL_PAPERS	58	112	217	438	219	444
PVM 3	101,000*	101,000	100,000	99,000	100,000	116,000

Table 5: Timing for ReduceAdd Operations (microseconds)

	8	16	32	64	Float	Double
MasPar MP-1	158	235	425	1,030	413	849
Paragon XP/S	700*	700*	700	700	710	690
PAPERS1	21	34	58	132	63	130
TTL_PAPERS	60	113	218	447	222	442
PVM 3	102,000*	102,000	101,000	105,000	103,000	138,000

Table 6: Timing for ReduceMul Operations (microseconds)

	8	16	32	64	Float	Double
MasPar MP-1	32	49	81	120	87	172
Paragon XP/S	690*	690*	690	690	710	690
PAPERS1	20	33	57	127	63	131
TTL_PAPERS	59	112	218	442	244	475
PVM 3	104,000	101,000	99,000	102,000	102,000	137,000

Table 7: Timing for ReduceMin Operations (microseconds)

	8	16	32	64	Float	Double
MasPar MP-1	28	40	64	131	72	140
Paragon XP/S	700*	700*	700	690	710	690
PAPERS1	20	33	57	127	63	131
TTL_PAPERS	59	113	218	439	297	458
PVM 3	104,000	101,000	98,000	98,000	99,000	140,000

Table 8: Timing for ReduceMax Operations (microseconds)

8. Bitwise Reduction Operations

`p_reduceAnd`*itype*`(d)`	Return the bitwise AND of d from each PE
`p_reduceOr`*itype*`(d)`	Return the bitwise OR of d from each PE

The TTL_PAPERS hardware communication mechanism is actually a 4-bit NAND of the data from all processors in the barrier group, which is accessed within the library by the in-line function `p_nand()`. Thus, a 4-bit bitwise OR can be implemented simply by complementing the inputs to the NAND. Likewise, a 4-bit bitwise AND is generated by complementing the result of a NAND. Similar logic is also used within PAPERS1.

Because bitwise operations are defined for each integer data type, there are nine different versions of each bitwise reduction operation. Fortunately, a 4**k*-bit bitwise operation can be implemented by performing *k* 4-bit operations and using shifts to disassemble and reassemble the values. The use of shifts is important because it ensures that the resulting value is computed correctly even though some of the machines in the cluster may differ in the byte order that they use to store data in local memory.

For example, the following C code defines the eight different multi-bit versions of `p_reduceOr`:

```
#define REDUCEOR(name, type, bitcnt) \
type \
name(register type datum) \
{ \
  if (last_mask & ~OUR_MASK) { \
    register int bits = (bitcnt - 4); \
 \
    /* Four bits at a time, or = nand not */ \
    do { \
      datum |= (p_nand((~(datum >> bits)) & 0xf) \
          << bits); \
    } while ((bits -= 4) >= 0); \
  } \
 \
  return(datum); \
}

REDUCEOR(p_reduceOr8, int8, 8)
REDUCEOR(p_reduceOr8u, uint8, 8)
REDUCEOR(p_reduceOr16, int16, 16)
REDUCEOR(p_reduceOr16u, uint16, 16)
REDUCEOR(p_reduceOr32, int32, 32)
REDUCEOR(p_reduceOr32u, uint32, 32)
REDUCEOR(p_reduceOr64, int64, 64)
REDUCEOR(p_reduceOr64u, uint64, 64)
```

Notice that the check involving `OUR_MASK` bypasses the entire communication if only this processor is in the current barrier group. Timing for AND and OR reductions are given in Table 9 and Table 10.

	8	16	32	64
MasPar MP-1	118	139	180	266
Paragon XP/S	700*	700*	700	700
PAPERS1	9	16	29	81
TTL_PAPERS	16	31	59	137
PVM 3	102,000	100,000	99,000	100,000

Table 9: Timing for ReduceAnd Operations (microseconds)

	8	16	32	64
MasPar MP-1	9.4	12	17	31
Paragon XP/S	710*	710*	710	710
PAPERS1	9	16	30	81
TTL_PAPERS	16	30	59	137
PVM 3	104,000	99,000	100,000	103,000

Table 10: Timing for ReduceOr Operations (microseconds)

9. Broadcast Operations

`p_bcastPut`*itype*`(d)` Broadcast put d from this PE
`p_bcastGet`*itype*`()` Broadcast get value from sending PE

Although both PAPERS1 and TTL_PAPERS directly implement broadcasting, these functions are actually implemented as macros of other PAPERS library routines. For example, both the TTL_PAPERS and PAPERS1 definitions of `p_bcastPut32()` and `p_bcastGet32()` are:

```
#define p_bcastPut32(d)   ((void) p_reduceOr32(d))
#define p_bcastGet32()    p_reduceOr32(0)
```

Notice that a PE which is simply getting the broadcast value is really contributing 0 to a global OR.

Table 11 summarizes the benchmark results for paired broadcast put and get operations. The MasPar times are very low because the global OR network is used (in the guise of the MPL `proc[]` construct) to obtain the value from the PE that is broadcasting and the SIMD broadcast hardware is then used to transmit that value to all PEs.

	8	16	32	64	Float	Double
Cray T3D (PVM)	?	?	?	82	?	82
MasPar MP-1	9.5	12	18	31	18	31
Paragon XP/S	210	220	210	210	210	210
PAPERS1	9	16	30	81	30	81
TTL_PAPERS	16	30	59	137	59	137
PVM 3	38,000	38,000	38,000	40,000	39,000	41,000

Table 11: Timing for Broadcast Put/Get Operations (microseconds)

10. Scan Operations

`p_scanAdd`*type*`(d)`	Return the sum of d up to this PE
`p_scanMul`*type*`(d)`	Return the product of d up to this PE
`p_scanAnd`*itype*`(d)`	Return the bitwise AND of d up to this PE
`p_scanOr`*itype*`(d)`	Return the bitwise OR of d up to this PE
`p_scanMin`*type*`(d)`	Return the minimum of d up to this PE
`p_scanMax`*type*`(d)`	Return the maximum of d up to this PE
`p_rank`*type*`(d)`	Return the rank of d for each PE, such that the smallest value has rank 0

A scan, or parallel prefix, computation is much like a reduction, but returns partial results to each of the processors. Like reductions, these operations can be performed in parallel or serially. For a small number of processors, using a gather operation to collect the data values in each processor and then serially computing the result for each processor in local memory works well, but there are many alternative implementations.

Aside from using a tree structured implementation (with the same barrier group numbering problem described for reductions in Section 7), many of the scan operations that use integer data can be implemented efficiently by methods based on `p_waitvec().` In some cases, the `p_waitvec()` would be used for processors to vote high or low relative to the value output by one processor (e.g., for a rank operation). For the `Bits` variants of the scans, a series of `p_waitvec()` operations can be used to transmit only the desired data bits from all processors.

Both TTL_PAPERS and PAPERS1 currently implement these operations using a gather and serial local evaluation. The timing of sum, product, AND, OR, minimum, and maximum parallel prefix operations are given in Tables 12 through 17. Although not strictly a scan operation, the ranking operation (used for sorting) behaves very much like a scan, and is thus benchmarked in Table 18.

	8	16	32	64	Float	Double
MasPar MP-1	381	561	920	1,640	1,041	1,799
Paragon XP/S	710*	700*	700*	700*	700*	700*
PAPERS1	35	55	91	197	97	204
TTL_PAPERS	63	124	227	463	245	456
PVM 3	96,000*	96,000*	98,000*	99,000*	98,000*	100,000*

Table 12: Timing for ScanAdd Operations (microseconds)

	8	16	32	64	Float	Double
MasPar MP-1	421	651	1,165	2,045	1,150	2,220
Paragon XP/S	710*	700*	700*	700*	700*	700*
PAPERS1	37	57	92	205	97	200
TTL_PAPERS	65	116	223	454	235	466
PVM 3	96,000*	96,000*	98,000*	99,000*	98,000*	100,000*

Table 13: Timing for ScanMul Operations (microseconds)

	8	16	32	64
MasPar MP-1	381	561	920	1,645
Paragon XP/S	710*	700*	700*	700*
PAPERS1	31	61	118	245
TTL_PAPERS	58	114	222	481
PVM 3	96,000*	96,000*	98,000*	99,000*

Table 14: Timing for ScanAnd Operations (microseconds)

	8	16	32	64
MasPar MP-1	381	561	920	1,640
Paragon XP/S	710*	700*	700*	700*
PAPERS1	34	68	131	272
TTL_PAPERS	67	120	244	501
PVM 3	96,000*	96,000*	98,000*	99,000*

Table 15: Timing for ScanOr Operations (microseconds)

	8	16	32	64	Float	Double
MasPar MP-1	390	573	934	1,662	940	1,740
Paragon XP/S	710*	700*	700*	700*	700*	700*
PAPERS1	36	56	92	198	104	206
TTL_PAPERS	61	114	220	442	226	465
PVM 3	96,000*	96,000*	98,000*	99,000*	98,000*	100,000*

Table 16: Timing for ScanMin Operations (microseconds)

	8	16	32	64	Float	Double
MasPar MP-1	390	571	934	1,663	941	1,741
Paragon XP/S	710*	700*	700*	700*	700*	700*
PAPERS1	36	56	92	198	102	206
TTL_PAPERS	61	114	220	443	225	451
PVM 3	96,000*	96,000*	98,000*	99,000*	98,000*	100,000*

Table 17: Timing for ScanMax Operations (microseconds)

	8	16	32	64	Float	Double
MasPar MP-1	5,465	6,754	9,318	14,451	9,884	14,975
Paragon XP/S	710*	700*	700*	700*	700*	700*
PAPERS1	37	57	93	201	103	205
TTL_PAPERS	63	116	222	510	228	522
PVM 3	96,000*	96,000*	98,000*	99,000*	98,000*	100,000*

Table 18: Timing for Rank Operations (microseconds)

11. Group Property Operations

`p_population()`	Return the total number of active PEs
`p_enumerate()`	Return a consecutive number for each active PE
`p_selectFirst()`	Return the lowest active PE number
`p_selectOne()`	Return any active PE's number

It is sometimes necessary to determine basic properties of the set of processors in the current barrier group. All of these functions compute properties directly from the local copy of the current barrier bit mask, and hence need no communication.

The value of `p_population()` is literally the population count of (number of 1 bits within) the barrier bit mask. Some machines make this operation a single instruction. For other machines, it can be implemented using a table to count bits within each chunk of the mask (e.g., an 8-bit mask section can be counted using a 256-element table). For very large clusters, the population count can be computed using a $\log_2 n$ time for n bits "SIMD bit summation" within each processor (e.g., for 4 PEs, `t=(mask&5)+((mask>>1)&5); population=(t>>2)+t;`).

The value of `p_enumerate()` can be determined in essentially the same way, but before computing the bit population, the barrier mask should be bitwise ANDed with the current processor's bit mask minus 1. Thus, each processor counts the number of participating processors in the mask that have lower processor numbers than itself.

The remaining two operations are actually implemented by just one function in the current PAPERS library, which simply scans the current barrier bit mask for the lowest 1 bit and returns that position. Again, some machines have single instructions that can perform this type of operation very efficiently, and table lookup also can be used.

The benchmark results are given in Table 19. Both PAPERS1 and TTL_PAPERS implement all of these functions as table lookups based on the barrier bit mask. The MasPar yields a more costly implementation because the bit mask is spread across the PEs (i.e., each PE knows only its own enable status), and communication is required. Because the other libraries do not support partitioning into arbitrary barrier groups, they have no barrier bit mask to examine, and these operations are not supported.

	Population	Enumerate	SelectFirst	SelectOne
MasPar MP-1	147*	572	35	12
PAPERS1	0.5	1	0.7	0.8
TTL_PAPERS	0.5	1	0.7	0.8

Table 19: Timing for Group Property Operations (microseconds)

12. Voting Operations

`p_voteCount(`*v*`)`	Return number of PEs that voted for us
`p_vote(`*v*`)`	Return vote bit mask
`p_matchCount`*type*`(`*v*`)`	Return number of PEs that voted as we did
`p_match`*type*`(`*v*`)`	Return match bit mask

One of the most powerful uses of the PAPERS library involves the use of PAPERS to statically schedule operations at runtime based on precise knowledge of the global state of the computation. We have found that such scheduling often is based on determining which PEs, or how many PEs, are contending for the same resource.

The most obvious resource upon which PEs might contend is the data bus that connects each PE to the PAPERS unit. For example, suppose that we need to have the sender, rather than the receiver, determine which PE will get a datum. Unlike putget, sender-targeted communication can have conflicts if multiple PEs want to send to the same PE. The receiver can determine that this contention exists by examining the return value obtained when each PE uses `p_voteCount()` to register the number of the PE to which it will send a datum. If the return value is 0, no PE targets this PE. If the return value is 1, the communication can be accomplished in one operation. A higher number indicates contention.

The use of `p_vote()` is similar to that of `p_voteCount()`, but the return value is a new barrier mask containing only processors that voted for this PE. Although this barrier mask might, or might not, be used to create a new barrier group, it can be used very effectively to statically schedule the contending accesses.

The concept of the match operations is very similar to that of the vote operations, but each PE gets either the count or the barrier mask representing the set of PEs that put forth the same data value as itself.

Although future versions of PAPERS will directly implement some of these functions, both PAPERS1 and TTL_PAPERS currently implement these operations by performing a sequence of $\log_2 n$ `p_waitvec()` operations to transmit an n-bit vote or match value from all PEs. The algorithm can be summarized as follows.

For each bit in the vote or match value, use `p_waitvec()` to transmit the bit. The returned bit vector is inverted if this PE's corresponding PE number/match value bit was a 0. By `and`ing all the resulting bit vectors together, we create a bit mask with a 1 in each position corresponding to a PE that is contending for/with us. The counts are obtained by simply performing a population count of bits in this mask.

The resulting execution times are listed in Table 20. Notice that none of the other libraries directly provides this type of scheduling support operation, although the MasPar's router network incorporates similar logic for detecting contention within its network.

	VoteCount	Vote	MatchCount8	Match8
PAPERS1	6.9	6.4	26	26
TTL_PAPERS	13	13	51	50

Table 20: Timing for Voting Operations (microseconds)

13. Conclusion

In this paper, we have presented a brief overview of the new communication model implemented by the PAPERS hardware and library. This library is suitable for use either as a vehicle for hand-coding applications or as a target for an optimizing compiler.

We have also documented that many of the functions provided by this library are familiar in the sense that similar functions appear in a variety of other libraries. However, as the detailed benchmarks reported in this paper show, the PAPERS library routines are consistently among the most efficient. We suggest that this efficiency is not due to our cleverness, but due to the use of a computation model which treats parallel communication as a tightly-coupled parallel operation rather than as a bunch of seemingly independent, potentially conflicting, block-oriented, high-latency, one-to-one transmissions.

The benefits of synchronous aggregate communication have long been known in the SIMD world, and have been cited as a primary advantage of SIMD/MIMD mixed-mode computation. However, it is a misconception that such operations require SIMD execution. Making aggregate operations efficient simply requires the ability to efficiently achieve synchronization across a group of PEs — which is to say it requires fast hardware barrier synchronization. One also has to be careful not to bury the communication hardware under too many latency-laden layers of interface software.

All aspects of the PAPERS system design and support software are fully public domain, and a wide variety of projects involving PAPERS are underway. For more information, see WWW URL:

`http://garage.ecn.purdue.edu/~papers/`

References

[BeS91] T.B. Berg and H.J. Siegel, "Instruction Execution Trade-Offs for SIMD vs. MIMD vs. Mixed Mode Parallelism," *5th International Parallel Processing Symposium*, April 1991, pp. 301-308.

[Cra93] Cray Research Incorporated, *Cray T3D System Architecture Overview*, Cray Research Incorporated, 1993.

[DiC95] H. G. Dietz, T. M. Chung, T. Mattox, and T. Muhammad, "Purdue's Adapter for Parallel Execution and Rapid Synchronization: The TTL_PAPERS Design," submitted to *ACM-IEEE-CS Supercomputing '95*, December 1995.

[DiS88] H. G. Dietz and T. Schwederski, *Extending Static Synchronization Beyond SIMD and VLIW*, Purdue University School of Electrical Engineering, Technical Report TR-EE 88-25, June 1988.

[GeS93] G. Geist and V. Sunderam, *Evolution of the PVM concurrent computing system*, 38th Annual IEEE Computer Society International Computer Conference, February 1993.

[Int94] Intel Corporation, *Paragon User's Guide*, Intel Supercomputer Systems Division, 1994.

[Mas91] MasPar Computer Corporation, *MasPar Programming Language (ANSI C compatible MPL) Reference Manual, Software Version 2.2*, Document Number 9302-0001, Sunnyvale, California, November 1991.

[Mes94] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Rice University, Technical Report CRPC-TR94439, April 1994.

[Muh95] T. Muhammad, *Hardware Barrier Synchronization For A Cluster Of Personal Computers*, Purdue University School of Electrical Engineering, MS Thesis, May 1995.

[OKD90] O'Keefe, M.T., and Dietz, H.G., Hardware barrier synchronization: static barrier MIMD (SBM). *Proc. of 1990 Int'l Conf. on Parallel Processing,* St. Charles, IL, August 1990, pp. I 35-42.

[OKD90a] O'Keefe, M.T., and Dietz, H.G., Hardware barrier synchronization: dynamic barrier MIMD (DBM). *Proc. of 1990 Int'l Conf. on Parallel Processing,* St. Charles, IL, August 1990, pp. I 43-46.