A Synchronization and Aggregate Communication Library for PAPERS Clusters

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

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

Abstract

PAPERS, Purdue's Adapter for Parallel Execution and Rapid Synchronization, is an inexpensive little box that can be plugged into four or more PCs or workstations to support use of these machines as a single, dedicated, parallel computer. Conventional clusters link UNIX-based workstations using a socket protocol over a local area network (LANs like Ethernet, FDDI, ATM, or HiPPI); this can yield substantial bandwidth, but cannot provide the low latency required for fine-grain parallel computation. Further, the block-oriented point-to-point operations provided by a LAN are not a good match for the aggregate communications commonly found in data parallel programs.

A PAPERS cluster consists of a PAPERS unit and a group of four or more unmodified PCs or workstations. Interactions between processors are accomplished by UNIX user processes directly accessing the PAPERS hardware through a standard parallel printer port; this limits the bandwidth somewhat, but bypasses the operating system and thus provides latencies as low as a few microseconds, including all hardware and software overheads. A typical PAPERS cluster can perform a barrier synchronization, a 4-bit global OR, or any of a variety of other aggregate operations, in just a few microseconds — no matter how many processors are in the cluster. Consequently, a PAPERS cluster can efficiently support fine-grain execution using a MIMD, SIMD, or VLIW model. Further, the PAPERS hardware design and support software is fully public domain, and PAPERS systems can be built easily and cheaply using standard components.

This paper details the PAPERS library for use with GCC.

The development of PAPERS was funded in part by Office of Naval Research (ONR) grant number N00014-91-J-4013. Facilities for the physical construction of the PAPERS prototypes were provided in part by National Science Foundation (NSF) award number CDA-9015696.

1. Introduction

PAPERS, Purdue's Adapter for Parallel Execution and Rapid Synchronizations, is designed to provide very low latency barrier synchronization and aggregate communication operations. Thus, the overriding concern in design and implementation of a public domain PAPERS support library has been to provide the most complete and useful library, while avoiding the usual software interface layering and the associated increases in latency. For this reason, the PAPERS library is based on the functionality of a very fine grain, proven effective, library: the library that supports MPL (a SIMD-parallel dialect of Gnu C) on MasPar's SIMD[1] supercomputers.

The reason that the PAPERS library isn't compliant with emerging standards for message-passing libraries (e.g., PVM or MPI) is simply that the basic structure of PAPERS operations is not an message-passing model. While interactions between PEs using message-passing systems are generally asynchronous and are always initiated by individual PEs, the PAPERS operations — and supporting library routines — are aggregate operations which are requested and performed in unison by all PEs (or 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 isn't an accident; years of research involving Purdue University's PASM (PArtitionable Simd Mimd) prototype supercomputer have experimentally demonstrated that a SIMD-like interaction is more efficient than traditional message-passing for a wide range of applications.

The basic model for parallel computation using the PAPERS library with GCC is outlined in Section 2. Section 3 provides an overview of how the library is organized. Each of Sections 4 through 11 describes the functionality, use, and implementation algorithms of a group of related functions. Most are accompanied by measured performance for the TTL_PAPERS implementation [DiC95]. To minimize the TTL_PAPERS hardware (e.g., a 4 processor unit uses just 8 TTL chips), direct support for some operations had to be removed; thus, the times quoted for the PAPERS library using TTL_PAPERS generally form the lower bound on PAPERS performance. Section 12 summarizes the contributions of this paper and directions for further research.

2. Parallel Computation Model

Since we have already mentioned that the functionality of the PAPERS library is based on the library that supports MPL on MasPar's SIMD supercomputers, it is natural to assume that our model for parallel computation is also based on SIMD execution semantics — however, this is not the case. In fact, our 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

[1] SIMD refers to Single Instruction stream, Multiple Data stream; i.e., a single processor with multiple function units organized so that the same operation can be performed simultaneously on multiple data values.

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 may look SIMD, but is in fact purely MIMD.

Suppose that, instead of executing just a barrier wait, all participating processors may additionally perform a communication operation. Again, the individual processors can be executing completely different programs. In fact, each can even behave differently within the communication (e.g., one processor broadcasts while others recieve). 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 wants this data, and to act accordingly in aggregate with A.

This model is most like synchronous messsage-passing in concept, but is much more powerful because the data values obtained can be any of a wide range of functions of the data from all processors within the aggregate. For example, reductions, scans (parallel prefix operations), and various other functions yield performance comparable to or better than the cost of performing a simple permutation communication. In summary, the PAPERS library model is pure barrier MIMD, but, because it efficiently implements aggregate communication operations, it can efficiently simulate SIMD or VLIW execution modes at a very fine grain level. The apparent execution mode also can be changed simply by changing the way in which the aggregate communication library routines are called.

3. PAPERS Library Overview

The PAPERS library discussed in this document is designed to provide all the basic components needed for writing real applications using the Gnu C Compiler. It is not the only PAPERS library, and it is still under construction in the sense that the underlying algorithms for some functions are being improved and there are still a variety of additions planned (e.g., both mirror and RAID file I/O routines are currently being developed).

Before reviewing each of the library routines, it is useful to review the way in which the library is structured. Although a typical user need not know where all the component pieces of the library come from, Section 3.1 provides an overview of which files define which functions. The PAPERS library routines directly handle a variety of different data types, so Section 3.2 describes what types are supported and how type information is incorporated into function names.

3.1. Library Organization

Although the PAPERS library incorporates many files, a casual user only needs to know about:

`paperslib.h`

This file declares all values and functions needed to use the PAPERS hardware. Simply including this file provides all the PAPERS library routines that a user-level program should ever need to access.

`paperslib.a`

The PAPERS library itself, including the linkable object code for all the support functions.

A more sophisticated user can choose to include declarations for only those portions of the library which are needed for each application. These files are:

`papers.h`

This file declares the minimum set of values and functions needed to use the PAPERS hardware. For example, like MasPar's library for MPL, the PAPERS library defines a variety of values relating to the number of processors available and the coordinates of this processor.

`gather.h`

This file contains the declarations of all the variants of gather operations, i.e., all functions that result in filling a local array with values gathered from each of the processors in the current barrier group.

`reduce.h`

This file contains the declarations of all the variants of reduction operations, i.e., all functions that reduce values from each of the processors in the current barrier group into a single value which is returned to all processors in the group.

`scan.h`

This file contains the declarations of all the variants of scan (parallel prefix) operations, i.e., all functions that produce result values that are a function not only of the data values from each of the processors in the current barrier group, but also of the processor numbers involved in the operation.

`putget.h`

This file contains the declarations of all the variants of putget (multibroadcast) operations, i.e., functions that allow each processor within the current barrier group to output a datum and input the datum that was output by the processor specified.

`timer.h`

This file contains the declarations of a few timer support functions. These are ordinary serial routines that simply provide a standard timing interface, despite the variations in timing facilities across different versions of UNIX.

`psound.h`

This file contains the declarations of a simple interface for making sounds using the PC speaker, primarily for debugging and demonstrations. Every PC has a speaker built-in, so listening to tones played simultaneously by a group of PCs within a cluster is convenient, and fine-grain timing anomalies are easily identified by audio effects (e.g., beating between different frequencies).

Still more sophisticated users may wish to performance-tweak some support library functions for a particular target machine (there is currently no assembly language code in any of the `.c` files), or may want to write additional PAPERS support functions. The following source files are available for modifying and rebuilding the library:

`papers.c`
: The source code for the PAPERS check-in, exit, and other basic operations.

`gather.c`
: The source code for the operations declared in `gather.h`.

`reduce.c`
: The source code for the operations declared in `reduce.h`.

`scan.c`
: The source code for the operations declared in `scan.h`.

`putget.c`
: The source code for the operations declared in `putget.h`.

`timer.c`
: The source code for the operations declared in `timer.h`.

`psound.c`
: The source code for the operations declared in `psound.h`.

`intern.h`
: In contrast to `papers.h`, this file defines all the implementation-specific values and creates in-line versions of the low-level functions needed to use the papers hardware. Because this file contains so many hardware-specific names, user programs should avoid including this file where naming conflicts may occur — all the user-level functionality is available through the cleaner interface of `papers.h`. However, this file should be included when defining your own new library routines or when absolute peak performance is needed.

`inline.h`
: This file contains the declarations of the low-level PAPERS routines that are typically used in-line within the rest of the PAPERS library. This file is included in other support files twice: once to create the in-line functions and once to generate ordinary externally-callable functions. This file should be modified only to add machine-specific low-level in-line functions.

Only `intern.h` and `inline.h` contain assembly code for the parallel port interface. Consequently, only these two files need to be modified if the PC/workstation hardware interface to the PAPERS unit is changed. However, switching from one version of PAPERS to another version may require changes throughout the entire library, because the aggregate operations implemented by hardware vary.

Of course, all these files are accompanied by a `Makefile` and a variety of simple test, demonstration, and benchmark programs.

3.2. Data Types

Unlike most implementations of PVM or MPI, the PAPERS hardware and interface software does not have a significant set-up time; an operation on *n* units of data generally takes 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 some PAPERS units is 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 (defined in the PAPERS `types.h` header file) and the function name suffix.

GCC Type	PAPERS Type	PAPERS Suffix
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, this is 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.

A similar portability constraint arrises 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.

4. PAPERS Management Operations (`papers.c`)

`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.

5. Low-Level PAPERS Operations (`inline.h`)

`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.

6. Gather Operations (`gather.c`)

`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 *n* broadcast operations, one from each of the *n* processors in the current barrier group. Because each processor in the group has a local copy of the barrier bit mask, 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.

Another aspect of gather implementation that might not be obvious is how the TTL_PAPERS hardware implements a broadcast using only 4-bit NAND communication. The answer is simply that if *n*-1 one bits are NANDed together with a single variable bit, the result is always the complement of the variable bit. Thus, the PEs that are not broadcasting are actually sending TTL_PAPERS a value of 0xf, while the PE that is broadcasting sends the complement of its 4-bit datum. The performance of this implementation is summarized in Table 2.

Function	Min Time μs	Avg Time μs	Max Time μs
p_gather8(p, d)	57	59.5	63
p_gather8u(p, d)	57	58	59
p_gather16(p, d)	113	113	113
p_gather16u(p, d)	115	115.5	117
p_gather32(p, d)	219	220.5	221
p_gather32u(p, d)	217	218.5	219
p_gather64(p, d)	449	449	449
p_gather64u(p, d)	477	508	527
p_gatherf(p, d)	239	247.5	255
p_gatherd(p, d)	441	443.5	447
p_gatherBits8u(p, d, 1)	11	12	13
p_gatherBits16u(p, d, 1)	11	12	13
p_gatherBits32u(p, d, 1)	11	11.5	13
p_gatherBits64u(p, d, 1)	13	14	15
p_gatherBits8u(p, d, 2)	19	19.5	21
p_gatherBits16u(p, d, 2)	19	21	23
p_gatherBits32u(p, d, 2)	19	19.5	21
p_gatherBits64u(p, d, 2)	21	23	25

Table 2: Typical 4-PE TTL_PAPERS Gather Timing

7. PutGet Operations (`putget.c`)

`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, TTL_PAPERS does not. Instead, it must map the putget into a series of simple broadcasts implemented by

NANDing (as described in Section 6). 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 innefficient 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.

Function	Min Time μs	Avg Time μs	Max Time μs
p_putget8(d, s)	59	62	67
p_putget8u(d, s)	87	110	125
p_putget16(d, s)	111	113	117
p_putget16u(d, s)	113	129	135
p_putget32(d, s)	215	217.5	219
p_putget32u(d, s)	217	218	219
p_putget64(d, s)	433	435.5	437
p_putget64u(d, s)	437	438.5	439
p_putgetf(d, s)	217	219	221
p_putgetd(d, s)	431	466.5	479
p_putgetBits8u(d, s, 1)	11	11.5	13
p_putgetBits16u(d, s, 1)	11	11.5	13
p_putgetBits32u(d, s, 1)	11	11	11
p_putgetBits64u(d, s, 1)	11	12	13
p_putgetBits8u(d, s, 2)	19	19.5	21
p_putgetBits16u(d, s, 2)	19	19	19
p_putgetBits32u(d, s, 2)	17	18.5	19
p_putgetBits64u(d, s, 2)	19	20.5	21

Table 3: Typical 4-PE TTL_PAPERS PutGet Timing

8. Arithmetic Reduction Operations (`reduce.c`)

`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 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 actually most efficient for a typical 4-processor TTL_PAPERS cluster. 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 `p_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).

Function	Min Time μs	Avg Time μs	Max Time μs
p_reduceAdd8(d)	59	59.5	61
p_reduceAdd8u(d)	59	59.5	61
p_reduceAdd16(d)	113	114	115
p_reduceAdd16u(d)	109	114.5	117
p_reduceAdd32(d)	219	222	223
p_reduceAdd32u(d)	223	223.5	225
p_reduceAdd64(d)	455	462.5	467
p_reduceAdd64u(d)	539	615.5	665
p_reduceAddf(d)	219	219	219
p_reduceAddd(d)	443	443.5	445
p_reduceAddBits8u(d, 1)	13	14	15
p_reduceAddBits16u(d, 1)	15	15.5	17
p_reduceAddBits32u(d, 1)	13	13	13
p_reduceAddBits64u(d, 1)	15	15	15
p_reduceAddBits8u(d, 2)	23	25.5	27
p_reduceAddBits16u(d, 2)	21	21.5	23
p_reduceAddBits32u(d, 2)	21	21.5	23
p_reduceAddBits64u(d, 2)	23	24.5	25

Table 4: Typical 4-PE TTL_PAPERS Addition Reduction Timing

Function	Min Time μs	Avg Time μs	Max Time μs
p_reduceMul8(d)	71	74.5	77
p_reduceMul8u(d)	61	61	61
p_reduceMul16(d)	119	119	119
p_reduceMul16u(d)	115	116.5	117
p_reduceMul32(d)	317	390	437
p_reduceMul32u(d)	221	222.5	223
p_reduceMul64(d)	449	450.5	451
p_reduceMul64u(d)	503	523.5	543
p_reduceMulf(d)	221	221.5	223
p_reduceMuld(d)	441	442	443
p_reduceMulBits8u(d, 1)	15	15	15
p_reduceMulBits16u(d, 1)	15	16	17
p_reduceMulBits32u(d, 1)	13	14	15
p_reduceMulBits64u(d, 1)	23	23.5	25
p_reduceMulBits8u(d, 2)	21	22.5	23
p_reduceMulBits16u(d, 2)	23	23	23
p_reduceMulBits32u(d, 2)	21	22	23
p_reduceMulBits64u(d, 2)	31	32	33

Table 5: Typical 4-PE TTL_PAPERS Multiplication Reduction Timing

Function	Min Time μs	Avg Time μs	Max Time μs
p_reduceMin8(d)	59	60.5	61
p_reduceMin8u(d)	61	62	63
p_reduceMin16(d)	113	114	115
p_reduceMin16u(d)	113	113.5	115
p_reduceMin32(d)	223	223.5	225
p_reduceMin32u(d)	221	221.5	223
p_reduceMin64(d)	449	453.5	455
p_reduceMin64u(d)	459	471	477
p_reduceMinf(d)	231	243.5	249
p_reduceMind(d)	461	475	481
p_reduceMinBits8u(d, 1)	15	15	15
p_reduceMinBits16u(d, 1)	15	15	15
p_reduceMinBits32u(d, 1)	13	14	15
p_reduceMinBits64u(d, 1)	17	18.5	19
p_reduceMinBits8u(d, 2)	23	24	25
p_reduceMinBits16u(d, 2)	23	23.5	25
p_reduceMinBits32u(d, 2)	21	21.5	23
p_reduceMinBits64u(d, 2)	25	26	27

Table 6: Typical 4-PE TTL_PAPERS Minimum Reduction Timing

Function	Min Time μs	Avg Time μs	Max Time μs
p_reduceMax8(d)	59	60.5	61
p_reduceMax8u(d)	61	61.5	63
p_reduceMax16(d)	113	114.5	115
p_reduceMax16u(d)	113	114.5	115
p_reduceMax32(d)	217	219.5	221
p_reduceMax32u(d)	209	223.5	229
p_reduceMax64(d)	437	444.5	447
p_reduceMax64u(d)	439	445	447
p_reduceMaxf(d)	273	297	317
p_reduceMaxd(d)	457	458	459
p_reduceMaxBits8u(d, 1)	15	15.5	17
p_reduceMaxBits16u(d, 1)	15	15	15
p_reduceMaxBits32u(d, 1)	15	15	15
p_reduceMaxBits64u(d, 1)	17	17.5	19
p_reduceMaxBits8u(d, 2)	25	25	25
p_reduceMaxBits16u(d, 2)	25	25.5	27
p_reduceMaxBits32u(d, 2)	21	22	23
p_reduceMaxBits64u(d, 2)	27	27	27

Table 7: Typical 4-PE TTL_PAPERS Maximum Reduction Timing

9. Bitwise Reduction Operations (`reduce.c`)

`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.

Because bitwise operations are defined for each integer data type, there are eight 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 all eight different 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.

Additionally, each of the bitwise reductions has a "`Bits`" version for each of the unsigned integer data types. These are similar to the above, except in that each function takes a second argument which specifies the total number of bits to transmit.

Function	Min Time μs	Avg Time μs	Max Time μs
p_reduceAnd8(d)	17	17.5	19
p_reduceAnd8u(d)	17	17.5	19
p_reduceAnd16(d)	33	33	33
p_reduceAnd16u(d)	31	32.5	33
p_reduceAnd32(d)	61	61.5	63
p_reduceAnd32u(d)	61	61.5	63
p_reduceAnd64(d)	139	139.5	141
p_reduceAnd64u(d)	141	141	141
p_reduceAndBits8u(d, 1)	11	11.5	13
p_reduceAndBits16u(d, 1)	9	10	11
p_reduceAndBits32u(d, 1)	9	9.5	11
p_reduceAndBits64u(d, 1)	9	10.5	11
p_reduceAndBits8u(d, 2)	15	16.5	17
p_reduceAndBits16u(d, 2)	17	17	17
p_reduceAndBits32u(d, 2)	17	17	17
p_reduceAndBits64u(d, 2)	17	18	19

Table 8: Typical 4-PE TTL_PAPERS Bitwise AND Reduction Timing

Function	Min Time μs	Avg Time μs	Max Time μs
p_reduceOr8(d)	17	18.5	19
p_reduceOr8u(d)	17	18.5	19
p_reduceOr16(d)	33	33	33
p_reduceOr16u(d)	33	34	35
p_reduceOr32(d)	61	61.5	63
p_reduceOr32u(d)	105	131.5	153
p_reduceOr64(d)	139	140	141
p_reduceOr64u(d)	139	140	141
p_reduceOrBits8u(d, 1)	9	10	11
p_reduceOrBits16u(d, 1)	9	10.5	11
p_reduceOrBits32u(d, 1)	9	10.5	11
p_reduceOrBits64u(d, 1)	11	11	11
p_reduceOrBits8u(d, 2)	17	17.5	19
p_reduceOrBits16u(d, 2)	17	18	19
p_reduceOrBits32u(d, 2)	17	17.5	19
p_reduceOrBits64u(d, 2)	17	18.5	19

Table 9: Typical 4-PE TTL_PAPERS Bitwise OR Reduction Timing

10. Scan Operations (`scan.c`)

`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 9), many of the scan operations that use integer data can be implemented efficiently by methods based on 1-bit multibroadcasts. In some cases, the 1-bit multibroadcast 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 1-bit multibroadcast operations can be used to transmit only the desired data bits from all processors.

Function	Min Time μs	Avg Time μs	Max Time μs
p_scanAdd8(d)	61	62.5	63
p_scanAdd8u(d)	65	66	67
p_scanAdd16(d)	123	124	125
p_scanAdd16u(d)	123	123.5	125
p_scanAdd32(d)	227	227	227
p_scanAdd32u(d)	231	232.5	233
p_scanAdd64(d)	447	462.5	469
p_scanAdd64u(d)	445	475	485
p_scanAddf(d)	237	244.5	251
p_scanAddd(d)	455	455.5	457

Table 10: Typical 4-PE TTL_PAPERS Addition Scan Timing

Function	Min Time μs	Avg Time μs	Max Time μs
p_scanMul8(d)	63	64.5	65
p_scanMul8u(d)	63	64	65
p_scanMul16(d)	115	116	117
p_scanMul16u(d)	115	115.5	117
p_scanMul32(d)	219	222.5	225
p_scanMul32u(d)	213	223.5	227
p_scanMul64(d)	445	453.5	457
p_scanMul64u(d)	441	473.5	485
p_scanMulf(d)	223	234.5	239
p_scanMuld(d)	453	465.5	471

Table 11: Typical 4-PE TTL_PAPERS Multiplication Scan Timing

Function	Min Time μs	Avg Time μs	Max Time μs
p_scanAnd8(d)	57	57.5	59
p_scanAnd8u(d)	57	57.5	59
p_scanAnd16(d)	113	113.5	115
p_scanAnd16u(d)	113	113	113
p_scanAnd32(d)	221	221.5	223
p_scanAnd32u(d)	225	226	227
p_scanAnd64(d)	481	494	499
p_scanAnd64u(d)	479	480.5	481
p_scanAndBits8u(d, 1)	9	10	11
p_scanAndBits16u(d, 1)	9	10.5	11
p_scanAndBits32u(d, 1)	9	9.5	11
p_scanAndBits64u(d, 1)	11	11	11
p_scanAndBits8u(d, 2)	17	18.5	19
p_scanAndBits16u(d, 2)	17	17	17
p_scanAndBits32u(d, 2)	17	17.5	19
p_scanAndBits64u(d, 2)	17	17.5	19

Table 12: Typical 4-PE TTL_PAPERS Bitwise AND Scan Timing

Function	Min Time μs	Avg Time μs	Max Time μs
p_scanOr8(d)	65	66.5	69
p_scanOr8u(d)	73	78	81
p_scanOr16(d)	119	119.5	121
p_scanOr16u(d)	123	124	125
p_scanOr32(d)	243	271	281
p_scanOr32u(d)	241	243.5	245
p_scanOr64(d)	507	519.5	527
p_scanOr64u(d)	491	500.5	505
p_scanOrBits8u(d, 1)	9	10	11
p_scanOrBits16u(d, 1)	11	12.5	13
p_scanOrBits32u(d, 1)	9	10.5	11
p_scanOrBits64u(d, 1)	11	11	11
p_scanOrBits8u(d, 2)	17	18	19
p_scanOrBits16u(d, 2)	17	18	19
p_scanOrBits32u(d, 2)	17	17.5	19
p_scanOrBits64u(d, 2)	17	19	21

Table 13: Typical 4-PE TTL_PAPERS Bitwise OR Scan Timing

Function	Min Time μs	Avg Time μs	Max Time μs
p_scanMin8(d)	71	79.5	85
p_scanMin8u(d)	59	60.5	61
p_scanMin16(d)	113	113.5	115
p_scanMin16u(d)	113	113.5	115
p_scanMin32(d)	219	219.5	221
p_scanMin32u(d)	219	220	221
p_scanMin64(d)	441	441.5	443
p_scanMin64u(d)	441	442	443
p_scanMinf(d)	225	226	227
p_scanMind(d)	447	464.5	471

Table 14: Typical 4-PE TTL_PAPERS Minimum Scan Timing

Function	Min Time μs	Avg Time μs	Max Time μs
p_scanMax8(d)	61	61	61
p_scanMax8u(d)	61	61	61
p_scanMax16(d)	113	114	115
p_scanMax16u(d)	113	114	115
p_scanMax32(d)	219	220	221
p_scanMax32u(d)	219	219.5	221
p_scanMax64(d)	441	442.5	443
p_scanMax64u(d)	445	445	445
p_scanMaxf(d)	225	225	225
p_scanMaxd(d)	449	450.5	451

Table 15: Typical 4-PE TTL_PAPERS Maximum Scan Timing

Function	Min Time μs	Avg Time μs	Max Time μs
p_rank8(d)	63	63	63
p_rank8u(d)	61	63.5	65
p_rank16(d)	157	180.5	203
p_rank16u(d)	115	115.5	117
p_rank32(d)	221	221.5	223
p_rank32u(d)	215	223.5	227
p_rank64(d)	483	510	535
p_rank64u(d)	499	560	595
p_rankf(d)	227	228	229
p_rankd(d)	495	521.5	543
p_rankBits8u(d, 1)	15	16	17
p_rankBits16u(d, 1)	15	16.5	17
p_rankBits32u(d, 1)	15	15.5	17
p_rankBits64u(d, 1)	17	18.5	19
p_rankBits8u(d, 2)	25	25	25
p_rankBits16u(d, 2)	23	24.5	25
p_rankBits32u(d, 2)	23	24	25
p_rankBits64u(d, 2)	27	27.5	29

Table 16: Typical 4-PE TTL_PAPERS Rank Ordering Timing

11. Group Counting Operations (typically in `scan.c`)

`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. 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.

Function	Min Time μs	Avg Time μs	Max Time μs
p_population()	0.5	0.5	0.5
p_enumerate()	0.5	0.5	0.5
p_selectFirst()	1	2	3
p_selectOne()	1	2.5	3

Table 17: Typical 4-PE TTL_PAPERS Group Counting Operation Timing

12. Conclusion

This paper has presented a brief, but complete, overview of how the PAPERS library is structured and how it performs using the minimum PAPERS hardware — TTL_PAPERS using a four-processor cluster of 486DX 33MHz machines.

This library will continue to grow and mature, however, it does support most basic application program needs. Expect major changes very soon:

- Addition of a mirrored parallel file I/O package.
- Addition of a RAID parallel file I/O package.
- A series of internal modifications to the check-in and check-out procedures, primarily designed to ensure that mutiple users cannot accidentally access the PAPERS hardware at the same time.
- Addition of user-level support for parallel interrupt handling.

Please consider this document as a draft only; it will be either updated or replaced as changes are made.

References

[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 *International Conference on Parallel Processing*, August 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.