PAPERS: Purdue's Adapter for Parallel Execution and Rapid Synchronization†

H. G. Dietz, T. Muhammad, J. B. Sponaugle, and T. Mattox

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

Abstract

There are a lot of 386/486/Pentium-based personal computers (PCs) out there. They are affordable, reliable, and offer good performance. Thus, it is only natural to think of networking multiple PCs to create a high-performance parallel machine — the problem is that conventional networking systems cannot provide low latency synchronization and communication. Low latency allows fine grain parallelism; the longer the latency, the fewer the programs that can achieve good speedup through use of parallelism.

Typical parallel machines constructed using PC networks (e.g., PVM software using Ethernet hardware) generally have latencies between 0.001s and 0.1s. Even the "best" commercially-available parallel computers can do no better than a latency corresponding to the time to execute hundreds to thousands of floating-point operations. In contrast, PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization) provides a latency corresponding to execution of just a few floating-point operations. Despite this, PAPERS can be implemented at a cost of less than $50/PC, including cables.

† 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 award number 9015696-CDA.

1. Theory of Operation

The reason PAPERS can achieve such low latency is that it is not a conventional network. Rather, it is a box implementing a special type of fine-grain barrier synchronization that facilitates compile-time scheduling of parallel operations [DiO92]: the full dynamic barrier functionality described in [OKD90a]. In fact, PAPERS implements communication only as a side-effect of barrier synchronization.

Hardware barrier synchronization was first proposed in a paper by Harry Jordon [Jor78], and has since become a popular mechanism for coordination of MIMD[1] parallel processes. A barrier synchronization is accomplished by processors executing a `wait` operation that does not terminate until sometime after all PEs have signaled that they are `waiting`. However, while building the 16 processor PASM (PArtitionable Simd Mimd) prototype in 1987 [SiN87], we realized that the hardware enabling a collection of conventional processors to execute both MIMD and and instruction-level SIMD[2] programs was actually an extended type of barrier synchronization mechanism. Generalizing this barrier synchronization mechanism resulted in several new classes of barrier synchronization architectures, as reported in [OKD90] [OKD90a]. The new barriers differ from previous concepts in that:

[1] Rather than requiring all processors to participate in every barrier, any arbitrary subset of the processors can participate in each barrier. This necessitates an efficient mechanism for enqueuing bit masks representing which processors participate in each barrier.

[2] The hardware ensures that all participating processors are allowed to proceed after a `wait` at precisely the moment that the last processor signals that it is `waiting`. It is this timing property that makes our barrier mechanism able to use conventional processors to efficiently implement fine-grain MIMD, SIMD, and VLIW[3] execution [CoD94a], as well as providing an efficient target for compile-time instruction-level code scheduling [DiO92].

[3] Because subsets of processors can participate in different barriers, it is possible to partition a parallel machine into smaller parallel submachines. Although partitioning is supported by all the mechanisms described in [OKD90] and [OKD90a], only the dynamic barrier functionality described in [OKD90a] efficiently supports arbitrary runtime partitioning.

The static barrier mechanism described in [OKD90] became popular almost instantly. However, the dynamic functionality of [OKD90a] required too much hardware to implement mask enqueuing and arbitrary partitioning.

---

[1] MIMD refers to Multiple Instruction stream, Multiple Data stream; i.e., each processor independently executes it own program, synchronizing and communicating with other processors whenever the parallel algorithm requires.

[2] 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.

[3] VLIW refers to Very Long Instruction Word; i.e., a generalization of SIMD that allows each function unit to perform a potentially different operation on its own data.

The final insight that allowed us to build PAPERS came late in 1993 [CoD94]. It redesigned the implementation of the dynamic functionality described in [OKD90a] so that fancy enqueuing logic and an associative mask matching memory were replaced by a very simple static barrier mechanism replicated for each processor. With this change, the enqueuing logic mutated into a communication mechanism that efficiently implements the simultaneous broadcast of one bit from each processor — very simple hardware, but very useful for general communications, as well as for determining future barrier masks.

Thus, PAPERS is the first implementation of the full dynamic barrier mechanism — it surely will not be the last. The current PAPERS prototype will no doubt be followed by a series of enhanced versions. The first prototype connects up to four PCs; later versions will be able to connect at least 16 PCs, and will provide greater communications bandwidth without increasing the latency.

Unlike most research prototype supercomputers, PAPERS is a fully public domain hardware and software design intended to be widely replicated.

The PAPERS systems will also serve as a software testbed for a prototype supercomputer based on the same barrier synchronization technology — CARDBoard, the Compiler-oriented Architecture Research Demonstration Board. CARDBoard differs from PAPERS in that it does not center on using PCs as processing elements, but simply uses PCs as hosts for up to 256 CARDBoards, each incorporating four high-performance RISC processors. Thus, CARDBoard will offer much higher performance than PAPERS (200 MFLOPS/board in the current design), but requires much more complex and specialized hardware and software.

2. PC Hardware

Although PAPERS provides very low latency synchronization and communication, it is interfaced to PCs using only a standard parallel printer port and is implemented with a minimal amount of external hardware. This section details the PC hardware involved in use of PAPERS.

Throughout the following description, we will distinguish between stand-alone PCs and PCs used as processors within a parallel machine by referring to the later as “PEs” — processing elements. The design presented here supports up to 4 PEs (PE0, PE1, PE2, and PE3); future modifications will scale the the design to larger configurations, probably up to 16 PEs.

2.1. PE Hardware Interface

No changes are required to make standard PC hardware into a PAPERS PE. All that is needed is a standard parallel printer port and an appropriate cable. Although some of the PCs on the market provide extended-functionality parallel ports that allow 8-bit bidirectional data connections, many PCs provide only an 8-bit data output connection. To ensure that PAPERS can be used with any PC, PAPERS uses only the functions supported by a standard unidirectional parallel port.

But if there is no parallel input port, how does PAPERS get data into the PC? The answer lies in the fact that the 8-bit data output port is accompanied by a variety of input and output status lines on two other ports associated with the 8-bit data output port. Counting these status lines, there are actually 12 bits of data output and 5 bits of data input.

The current PAPERS design uses 11 of the 12 available output lines and all 5 of the input lines. The pin/contact assignment for each of these lines is given in Tables 1 and 2. Table 1 lists the pin numbers as they appear on the PE’s DB25 connector. Table 2 lists the contact numbers for the signals as they appear on the PAPERS’ 36-pin Centronics connector.

Table 1: DB25 Parallel Port Pin Assignments

Pin #	Std. Name	Use In PAPERS
Pin 1	Strobe	U0 (User bit 0)
Pin 2	D0	D (Data Bit Value)
Pin 3	D1	
Pin 4	D2	IR (Interrupt Request)
Pin 5	D3	S (Barrier Sync. Request)
Pin 6	D4	B0 (Barrier Mask Contains PE0)
Pin 7	D5	B1 (Barrier Mask Contains PE1)
Pin 8	D6	B2 (Barrier Mask Contains PE2)
Pin 9	D7	B3 (Barrier Mask Contains PE3)
Pin 10	Ack	INT (Interrupt)
Pin 11	Busy	GO (Barrier Sync. Completed)
Pin 12	PE	I2 (PExI2 = PEyD such that y=(x+3)%4)
Pin 13	SlctIn	I1 (PExI1 = PEyD such that y=(x+2)%4)
Pin 14	AutoFD	U1 (User bit 1)
Pin 15	Error	I0 (PExI0 = PEyD such that y=(x+1)%4)
Pin 16	Init	GI (GO Causes Interrupt)
Pin 17	Slct	CE (Connection Established)
Pin 18	Gnd	
Pin 18	Gnd	
Pin 19	Gnd	
Pin 20	Gnd	
Pin 21	Gnd	
Pin 22	Gnd	
Pin 23	Gnd	
Pin 24	Gnd	
Pin 25	Gnd	

Table 2: Centronics Connector Contact Assignments

Contact #	Std. Name	Use In PAPERS
Contact 1	Strobe	U0 (User bit 0)
Contact 2	D0	D (Data Bit Value)
Contact 3	D1	
Contact 4	D2	IR (Interrupt Request)
Contact 5	D3	S (Barrier Sync. Request)
Contact 6	D4	B0 (Barrier Mask Contains PE0)
Contact 7	D5	B1 (Barrier Mask Contains PE1)
Contact 8	D6	B2 (Barrier Mask Contains PE2)
Contact 9	D7	B3 (Barrier Mask Contains PE3)
Contact 10	Ack	INT (Interrupt)
Contact 11	Busy	GO (Barrier Sync. Completed)
Contact 12	PE	I2 (PExI2 = PEyD such that y=(x+3)%4)
Contact 13	SlctIn	I1 (PExI1 = PEyD such that y=(x+2)%4)
Contact 14	AutoFD	U1 (User bit 1)
Contact 19	Ground	
Contact 20	Ground	
Contact 21	Ground	
Contact 22	Ground	
Contact 23	Ground	
Contact 24	Ground	
Contact 25	Ground	
Contact 31	Init	GI (GO Causes Interrupt)
Contact 32	Error	I0 (PExI0 = PEyD such that y=(x+1)%4)
Contact 36	Slct	CE (Connection Established)

2.2. PE Port Bit Assignments

Although the parallel port hardware is not altered to work with PAPERS, the parallel port lines are not used as they would be for driving a Centronics-compatible printer. Thus, it is necessary to replace the standard parallel port driver software with a driver designed to interact with PAPERS. Toward this end, it is critical to understand which port addresses, and bits within the port registers, correspond with each PAPERS signal.

There are three port registers associated with a PC parallel port. These ports have I/O addresses corresponding to the port base address (henceforth, called “PortBase”) plus 0, 1, or 2.

Typically, PortBase will be one of 0x378, 0x278, or 0x3bc, corresponding to MS-DOS printer names LPT1:, LPT2:, and LPT3:. Check the documentation for your PC system to determine the appropriate PortBase value for the parallel port that you are using as the interface to PAPERS.

The bit assignments for the first port register, PortBase + 0, are listed in Table 3. This register is used to send PAPERS the information used in each barrier synchronization. Notice that bit 1 is currently unassigned, but should be set to 0.

Table 3: PortBase + 0 Bit Assignments

Bit	Name	Use In PAPERS
bit 7	D7	B3 (Barrier Mask Contains PE3)
bit 6	D6	B2 (Barrier Mask Contains PE2)
bit 5	D5	B1 (Barrier Mask Contains PE1)
bit 4	D4	B0 (Barrier Mask Contains PE0)
bit 3	D3	S (Barrier Sync. Request)
bit 2	D2	IR (Interrupt Request)
bit 1	D1	0 (reserved for future use)
bit 0	D0	D (Data Bit Value)

The second port register, PortBase + 1, is used to receive information from PAPERS. Bit assignments for this register are given in Table 4. The arrangement of bits within this register is the result of the fact that PCs usually can generate an interrupt signal when Ack is set; the interrupt line must be the Ack signal. The three remaining contiguous bits of the register are thus designated as the data input from other PEs. This leaves bit 7 as the GO signal — the bit tested to determine if synchronization has been achieved. It happens that the sense of bit 7 is inverted on the port; the PAPERS hardware compensates for this so that a port read sees the GO bit as a 1 when the barrier has fired.

Table 4: PortBase + 1 Bit Assignments

Bit	Name	Use In PAPERS
bit 7	Busy	GO (Barrier Sync. Completed)
bit 6	Ack	INT (Interrupt)
bit 5	PE	I2 (PExI2 = PEyD such that y=(x+3)%4)
bit 4	SlctIn	I1 (PExI1 = PEyD such that y=(x+2)%4)
bit 3	Error	I0 (PExI0 = PEyD such that y=(x+1)%4)
bit 2	unused	
bit 1	unused	
bit 0	unused	

The third port register, PortBase + 2, is used by PAPERS only for output bits that change value relatively rarely — the software does not access this register in the course of executing a typical barrier synchronization. In other words, this register is used for the "modal" information outlined in Table 5. Although this discussion refers to the signals as they are listed in Table 5, the port actually inverts the sense of bits 3, 2, 1, and 0; compensation for this inversion is done (by XOR with 0xf) inside the lowest-level PAPERS port driver.

Table 5: PortBase + 2 Bit Assignments

Bit	Name	Use In PAPERS
bit 7	unused	
bit 6	unused	
bit 5	unused	
bit 4	IntEn	IE (Interrupt Enable)
bit 3	Slct	CE (Connection Established)
bit 2	Init	GI (GO Causes Interrupt)
bit 1	AutoFD	U1 (User bit 1)
bit 0	Strobe	U0 (User bit 0)

Part of the modal information involves the control of interrupts. Normal operation of PAPERS does not require interrupts. However, PAPERS does support the use of interrupts for two separate types of events:

[1] If the operating system on one PE determines that the other PEs should be informed of some event, it can interrupt *any subset* or *all* of the PEs by setting its IR bit (see Table 3). If any PE asserts its IR bit, then the INT bit (see Table 4) of each PE in this PE's barrier mask will be 1 and the GO bit (see Table 4) will be 0. Notice that the PE setting its IR bit will only interrupt itself if the corresponding bit is on in its barrier mask.

[2] If desired, the PAPERS barrier hardware can be set to interrupt this PE when PAPERS determines that this PE's barrier has been satisfied. This response can be independently enabled/disabled by each PE by setting its GI bit. If GI is 1, then completion of a barrier is signaled by setting both the INT and GO bits to 1. If GI is 0, then only the IR bit can cause INT to be 1.

The setting of the INT bit (see Table 4) happens whether the PE has enabled interrupts or not; an actual processor interrupt occurs only if the IE bit is set to 1. Use of true processor interrupts can be problematic due to conflicts with other devices sharing the same interrupt vector (e.g., the same interrupt is often generated by both the parallel port and a sound card). Thus, it may be preferable for PEs to use polling to detect these "interrupt" conditions.

There are three other modal bits in port PortBase + 2. These bits are not actually used by the logic in PAPERS, but rather are used to drive an informational status display. The CE bit is used to indicate that the PAPERS hardware has been properly connected to the PE. The other two bits are "user-defined status" bits that can be used in any way desired, however, the suggested use is to encode the function that the PAPERS hardware is being used to implement. This use is summarized in Table 6.

Table 6: Meaning Of U1 And U0 Signals

U1	U0	Meaning
0	0	PAPERS is not currently in use
0	1	PAPERS is being used to barrier synchronize
1	0	PAPERS is being used to transmit user data
1	1	PAPERS is being used by the operating system

3. PAPERS Hardware

Thus far, this document has focussed on the way in which PC hardware interacts with PAPERS. In this section, we briefly describe the hardware that implements PAPERS itself. Notice that there is, in fact, very little hardware inside PAPERS — which is why PAPERS is so inexpensive to build and fast to operate.

3.1. Logic Design

The active part of the PAPERS hardware follows the basic design presented in [CoD94], and is implemented using four small PLAs (programmable logic arrays) — one for each PE. For each PE, only two signals are actively derived from the signals fed into the PAPERS box: the GO and INT signals (see Table 4). The other three input signals for each PE (I2, I1, and I0) are literally wires directly connected to the D outputs of the other PEs.

Although the four PLAs are connected differently, their internal logic is the same and the differences in the connections follow a simple pattern. The following description refers to the PLA for PEa with respect to PEb, PEc, and PEd. Given a PE number for a, the PE numbers for b, c, and d can be derived by: b=(a+1)%4, c=(a+2)%4, d=(a+3)%4. Externally, each PLA appears as shown in Figure 1.

Figure 1: PLA Pin Layout

PAL22V10

CLK	1	24	VCC
PEaBb	2	23	PEaBa
PEaBc	3	22	GO
PEaBd	4	21	PEaGI
PEbBa	5	20	PEaGO
PEcBa	6	19	PEdS
PEdBa	7	18	PEcS
PEaIR	8	17	PEbS
PEbIR	9	16	CLKRST
PEcIR	10	15	CLKSET
PEdIR	11	14	PEaINT
GND	12	13	PEaS

Most of the PLA's input signals are taken directly from the PE parallel ports, however, there are a few surprises. The basic barrier logic tree described in [CoD94] derives the CLKSET signal — it does not directly derive PEaGO or even GO. This is because when the barrier becomes satisfied we want to latch a 1 bit into GO; thus, 1 is hardwired on the input to the GO register and the CLKSET signal is used as a clock to cause 1 to be sampled. This use requires CLKSET to be externally wired to the CLK input. When all PEs involved in a synchronization have read their input data, the (asynchronous) reset of the GO register is internally triggered by CLKRST.

The difference between the internal GO register and the PEaGO output signal is that interrupts can change the meaning of the GO bit. In essence, the PEaGO and PEaINT signals are really encoding a two bit PAPERS hardware state, as outlined in Table 7.

Table 7: Meaning Of PEaGO And PEaINT Signals

PEaGO	PEaINT	Meaning
0	0	No interrupt, synchronization not achieved
1	0	No interrupt, synchronization achieved
0	1	Interrupt for all PEs (given priority over achieving synchronization)
1	1	Interrupt for synchronization achieved

The PALASM code for the PLA follows. No attempt has been made to simplify these equations, since PALASM automatically minimizes the PLA complexity.

```
;---------------------------------- PIN Declarations ---------------

PIN     1       CLK     COMBINATORIAL
PIN     2       PEaBb   COMBINATORIAL
PIN     3       PEaBc   COMBINATORIAL
PIN     4       PEaBd   COMBINATORIAL
PIN     5       PEbBA   COMBINATORIAL
PIN     6       PEcBa   COMBINATORIAL
PIN     7       PEdBa   COMBINATORIAL
PIN     8       PEaIR   COMBINATORIAL
PIN     9       PEbIR   COMBINATORIAL
PIN     10      PEcIR   COMBINATORIAL
PIN     11      PEdIR   COMBINATORIAL
PIN     12      GND
PIN     13      PEas    COMBINATORIAL
PIN     14      PEaINT  COMBINATORIAL
PIN     15      CLKSET  COMBINATORIAL
PIN     16      CLKRST  COMBINATORIAL
PIN     17      PEbs    COMBINATORIAL
PIN     18      PEcs    COMBINATORIAL
PIN     19      PEds    COMBINATORIAL
PIN     20      PEaGO   COMBINATORIAL
PIN     21      PEaGI   COMBINATORIAL
PIN     22      GO      REGISTERED
PIN     23      PEaBa   COMBINATORIAL
PIN     24      VCC
NODE    1       GLOBAL
```

```
;---------------------------------- Boolean Equation Segment ------
EQUATIONS

PEaINT = ((PEaIR*PEaBa) +
          (PEbIR*PEbBa) +
          (PEcIR*PEcBa) +
          (PEdIR*PEdBa) +
          (GO*PEaGI))

GO = VCC

GLOBAL.RSTF = CLKRST

CLKSET = ((/PEaBa+(PEaBa*PEaS)) *
          (/PEaBb+(PEbBa*PEbS)) *
          (/PEaBc+(PEcBa*PEcS)) *
          (/PEaBd+(PEdBa*PEdS)) *
          /GO)

CLKRST = ((/PEaBa+(PEaBa*/PEaS)) *
          (/PEaBb+(PEbBa*/PEbS)) *
          (/PEaBc+(PEcBa*/PEcS)) *
          (/PEaBd+(PEdBa*/PEdS)) *
          GO)

/PEaGO = ((PEaINT*/((PEaIR*PEaBa) +
                    (PEbIR*PEbBa) +
                    (PEcIR*PEcBa) +
                    (PEdIR*PEdBa))) +
          (/PEaINT*GO))
```

Notice that, if desired, the entire PAPERS design also could be implemented either by a single larger PLA or by a manageable number of simple gate-level chips. We suggest that using one or more larger PLAs (e.g., Xilinx parts) is probably the most effective way to scale PAPERS to handle larger numbers of PEs.

3.2. Packaging

The prototype PAPERS unit is housed in a natural finish red oak box that is 11.75" wide by 6" deep by 6" tall, with a simple 3" steel handle protruding by 1" on the left side (to aid in carrying the system for demonstrations at remote sites). Inside the box, on the left side there is one 4" by 6" wire-wrapped card containing the PLAs and LED driving circuitry; on the right side is a 5 volt switching power supply (although a maximum of less than 1.5 amps is needed, we used a supply rated at 3 amps). The cover of the case is a simple piece of 0.25" thick oak, attached by velcro and perforated above the power supply to allow convection cooling. Behind the circuit card on the back of the box are four panel-mounted Centronics connectors — so that the cables used to connect PEs to PAPERS are standard PC parallel printer cables. The AC cord enters the box from behind the power supply. In front of the circuit card, rear-mounted on the front panel, is an array of LEDs used as a status display for the PEs connected to PAPERS.

Strictly speaking, there is no need to have any display connected to the PAPERS hardware. Indeed, eliminating the display can greatly simplify the hardware because it eliminates the need for LED drivers and perhaps even eliminates the separate power supply (the PLAs might be powered by the parallel port, but there is not enough power to drive the LEDS). However, PAPERS is a research prototype: the LEDs make it a lot easier to see what is happening... and to debug the system.

The prototype LED display consists of 40 LEDs arranged in 4 columns, each column representing the status of one PE. These columns are numbered in decreasing order from left to right (as the LEDs are normally viewed), i.e., PE3 PE2 PE1 PE0. The signal descriptions are given in Table 6. Notice that none of the LEDs displays a derived signal — this is because the two derived signals change value only momentarily, so fast that the state change would not be perceptible.

Display Position	Label On The PAPERS Unit	PAPERS Signal
Green LED 5	PE Connection Established	CE
Green LED 4	Interrupt Request	IR
Green LED 3	User-defined Status Bit 1	U1
Green LED 2	User-defined Status Bit 0	U0
Green LED 1	Data Bit Value	D
Green LED 0	Barrier Sync. Request	S
Amber LED 3	Barrier Mask Contains PE3	B3
Amber LED 2	Barrier Mask Contains PE2	B2
Amber LED 1	Barrier Mask Contains PE1	B1
Amber LED 0	Barrier Mask Contains PE0	B0

Of course, there are some active components used for more than driving LEDs. The critical portion of the current version of PAPERS is implemented by four identical PLAs. Although, in theory, Figure 1 combined with the rules given earlier suffices to completely specify how the PLAs are connected to the PE signals, it isn't exactly easy to see how the chips get wired. Thus, Figure 2 gives the pinouts for all four PLAs.

Figure 2: Specific Pin Assignments for All Four PLAs

PE0 PLA

CLK	1	24	VCC
PE0B1	2	23	PE0B0
PE0B2	3	22	GO
PE0B3	4	21	PE0GI
PE1B0	5	20	PE0GO
PE2B0	6	19	PE3S
PE3B0	7	18	PE2S
PE0IR	8	17	PE1S
PE1IR	9	16	CLKRST
PE2IR	10	15	CLKSET
PE3IR	11	14	PE0INT
GND	12	13	PE0S

PE1 PLA

CLK	1	24	VCC
PE1B2	2	23	PE1B1
PE1B3	3	22	GO
PE1B0	4	21	PE1GI
PE2B1	5	20	PE1GO
PE3B1	6	19	PE0S
PE0B1	7	18	PE3S
PE1IR	8	17	PE2S
PE2IR	9	16	CLKRST
PE3IR	10	15	CLKSET
PE0IR	11	14	PE1INT
GND	12	13	PE1S

PE2 PLA

CLK	1	24	VCC
PE2B3	2	23	PE2B2
PE2B0	3	22	GO
PE2B1	4	21	PE2GI
PE3B2	5	20	PE2GO
PE0B2	6	19	PE1S
PE1B2	7	18	PE0S
PE2IR	8	17	PE3S
PE3IR	9	16	CLKRST
PE0IR	10	15	CLKSET
PE1IR	11	14	PE2INT
GND	12	13	PE2S

PE3 PLA

CLK	1	24	VCC
PE3B0	2	23	PE3B3
PE3B1	3	22	GO
PE3B2	4	21	PE3GI
PE0B3	5	20	PE3GO
PE1B3	6	19	PE2S
PE2B3	7	18	PE1S
PE3IR	8	17	PE0S
PE0IR	9	16	CLKRST
PE1IR	10	15	CLKSET
PE2IR	11	14	PE3INT
GND	12	13	PE3S

4. PAPERS Software

Although PAPERS will be supported by a variety of software tools including public domain compilers for parallel dialects of both C and Fortran [DiO92] [CoD94a], in this document we restrict our discussion to the most basic hardware-level interface. The code given is written in C (the ANSI C-based dialect accepted by GCC) and is intended to be run under a unix-derived operating system. However, this interface software can be adapted to most existing (sequential) language compilers and interpreters under nearly any operating system.

The following sections discuss the operating system interface, PAPERS port access, and a simple barrier interface.

4.1. Operating System Interface

Although it would certainly be possible to implement the PAPERS software interface as part of an operating system's kernel, it is more efficient for an ordinary user program to directly access the ports connected to the PAPERS hardware. Although the ports can be directly accessed under most operating systems, here we focus on what it takes for a program under generic UNIX or Linux to gain port access.

4.1.1. Generic UNIX

In general, UNIX allows user processes to have direct access to all I/O devices. However, only processes that have a sufficiently high I/O priority level can make such accesses. Further, only a priviledged process can increase its I/O priority level — by calling `iopl()`. The following C code suffices:

```
if (iopl(3)) {
    /* iopl failed, implying we were not priv */
    exit(1);
}
```

But beware! This call grants the user program access to *all* I/O, including a multitude of unrelated ports.

In fact, this call allows the process to execute instructions enabling and disabling interrupts. By disabling interrupts, it is possible to ensure that all processors involved in a barrier synchronization act precisely in unison; thus, the number of port operations (barrier synchronizations) needed to accomplish PAPERS operations can be dramatically reduced. A basic barrier synchronization takes at least four port operations when timing cannot be ensured, but only two port operations with interrupts disabled. However, background scheduling of DMA devices (e.g., disks) and other interference makes it hard to be sure that a unix will provide precise timing constraints even when interrupts are disabled, so we do not advocate disabling interrupts.

Even so, performance of the barrier hardware can be safely improved by causing unix to give priority to a process that is waiting for a barrier synchronization. This improves performance because if any one PE is off running a process that has nothing to do with the synchronization, then all PEs trying to synchronize with that PE will be delayed. The priority of a priviledged unix process can increased by a call like:

```
/* set priority just below critical OS code */
nice(-20);
```

The argument to `nice()` should be a negative value between -20 and -1.

4.1.2. Linux

Although Linux supports the unix interface described in the previous section, it also provides a more secure way to obtain access to the I/O devices. The `ioperm()` function allows a priviledged process to obtain access to only the specified port or ports. The C code:

```
if (ioperm(PortBase, 3, 1)) {
    /* like iopl, failure implies we were not priv */
    exit(1);
}
```

Would obtain access for 3 ports starting at a base port address of PortBase.

Because the 386/486/Pentium hardware checks port permissions, this security does not destroy port I/O performance; however, checking the permission bits does add some overhead. For a typical PC parallel printer port, the additional overhead is just a few percent, and is probably worthwhile for user programs.

4.2. Port Access

Although Linux and most versions of unix provide routines for port access, these routines often provide a built-in delay loop to ensure that port states do not change faster than the external device can examine the state. Consequently, the PAPERS support code uses its own direct assembly language I/O calls. The code is:

```
inline unsigned int
inb(unsigned short port)
{
    unsigned char _v;
    delay();
__asm__ __volatile__ ("inb %w1,%b0"
        :"=a" (_v)
               :"d" (port), "0" (0));
    return _v;
}
```

```
inline void
outb(unsigned char value,
unsigned short port)
{
    delay();
__asm__ __volatile__ ("outb %b0,%w1"
        : /* no outputs */
        :"a" (value), "d" (port));
}
```

However, these port I/O calls are “sanitized for your protection” by the following PAPERS-specific macro definitions. Notice that the modal port inverts its four output lines; the `P_MODE()` macro includes an exclusive or operation to compensate for this. The code is:

```
/*      Stuff concerning the regular output port...
*/
#define P_OUT(x) \
        outb(((unsigned char)(x)), ((unsigned short) PortBase))
#define B3      0x80            /* Barrier Mask Contains PE3 */
#define B2      0x40            /* Barrier Mask Contains PE2 */
#define B1      0x20            /* Barrier Mask Contains PE1 */
#define B0      0x10            /* Barrier Mask Contains PE0 */
#define S       0x08            /* Barrier Sync. Request */
#define IR      0x04            /* Interrupt Request */
#define D       0x01            /* Data Bit Value */

/*      Stuff concerning the input port...
*/
#define P_IN() \
        inb((unsigned short) (PortBase + 1))
#define GO      0x80            /* Barrier Sync. Completed */
#define INT     0x40            /* Interrupt */
#define I2      0x20            /* PExD such that x=(iproc+3)&3 */
#define I1      0x10            /* PExD such that x=(iproc+2)&3 */
#define I0      0x08            /* PExD such that x=(iproc+1)&3 */
```

```
/*      Stuff concerning the modal output port...
*/
#define P_MODE(x) \
        outb(((unsigned char)((x) ^ (U0 | U1 | GI | CE)), \
             ((unsigned short)(PortBase + 2)))
#define IE      0x10            /* Interrupt Enable */
#define CE      0x08            /* Connection Established */
#define GI      0x04            /* Go Causes Interrupt */
#define U1      0x02            /* User bit 1 */
#define U0      0x01            /* User bit 0 */
```

4.3. Barrier Interface

The basic barrier interface consists of C `inline` routines serving three primary functions. The first is called `enqueue()`, and is used to enqueue a new barrier mask that will be used by all barrier operations until a new mask is enqueued. The second function is performed by `p_wait()`. It barrier synchronizes using the current barrier mask. The third is much like the second, but also communicates one bit of data from each PE. It is called `p_waitvec()`, because it barrier synchronizes and then returns a vector constructed using one bit from each PE.

4.3.1. Barrier `enqueue()`

The barrier enqueue operation consists primarily of sending PAPERS the requested barrier pattern, however, there is more to this than one might expect. One detail is that if there is no change in the mask, we have nothing to do. Another detail is that the mask pattern cannot be changed until everyone who was synchronizing with it is done reading their data. This is because our D bit is not buffered for each PE that might read it, but is directly examined by other PEs. Finally, there is the detail that `enqueue()` should set special variables so that other routines can cheaply obtain the current barrier mask as either the mask itself, `last_mask`, or as the mask shifted for `P_OUT()`, `last_pout`. The code is:

```
extern inline void
enqueue(register barrier mask)
{
        /* If appropriate, enqueue the new barrier pattern given by
           mask.  Note that mask represents PEk by bit k -- not quite
           the way that the port hardware works.
        */
```

```
#ifdef  PARANOID
        if (mask & ~ALL_MASK) {
                /* Bits are on for nonexistant PEs */
                p_error("enqueue of barrier containing nonexistant PE");
        }
#endif

        /* Is mask different from what we had? */
        if (mask != last_mask) {
                /* If last_mask had somebody beside us... */
                if (last_mask & ~OUR_MASK) {
                        /* Wait for barrier reset */
                        while (P_IN() & GO);
                }

                /* Update mask info & enqueue mask with PAPERS */
                P_OUT(last_pout = ((last_mask = mask) << 4));
        }
}
```

4.3.2. Barrier `p_wait()`

The basic barrier wait operation is `p_wait()`, as coded below. However, there is a minor optimization in that there is no need to use the PAPERS hardware to synchronize with yourself. Thus, there are two minor variants of this operation. Ordinary user code would use `p_wait()`, however, `_p_wait()` is used within library routines where we already know that we are synchronizing with at least one other PE.

```
extern inline void
_p_wait(register portdata p)
{
        /* Simple barrier synchronization on p. */

        while (P_IN() & GO);     /* wait for barrier reset */
        P_OUT(p | S);            /* synchronize on p */
        while (!(P_IN() & GO)); /* wait for barrier GO signal */
        P_OUT(p);                /* reset barrier */
}
```

```
extern inline void
p_wait()
{
        /* Simple barrier synchronization on current barrier.
           Nothing to do unless we are not alone...
        */
        if (last_mask & ~OUR_MASK) {
                _p_wait(last_pout);
        }
}
```

4.3.3. Barrier `p_waitvec()`

The `p_waitvec()` and `_p_waitvec()` routines are very similar to the `p_wait()` and `_p_wait()` routines, however, these routines transmit one bit of data from each PE as a side-effect of the barrier synchronization. There are four new complications introduced by this data transmission.

The first is that the bit transmitted by this PE is not made available to this PE through the `P_IN()` value. Thus, this bit must be determined directly from the local variable, as seen in `p_waitvec()`.

A second problem lies in the fact that the GO signal is better driven than the data bits. This can result in a slight delay between the `P_IN()` GO bit becoming 1 and the data bits achieving their final values. This is remedied within `_p_waitvec()` by simply issuing one more `P_IN()` after the GO bit has become 1. Future versions of PAPERS will hopefully correct this problem in hardware.

The third complication is that the data bits that correspond to each PE in the `P_IN()` value are not positioned such that PE*k* is represented by bit *k*. This problem is solved using a PE-dependent lookup table to re-map the bits to their standard positions.

Finally, there is the problem that all input bits have values, but only bits corresponding to PEs that synchronized with us are valid. Thus, `p_waitvec()` ensures that the bits for all PEs that did not participate in the barrier are set to 0.

Including the handling of these problems, the code is:

```
extern inline portdata
_p_waitvec(register portdata b)
{
        /* Simple barrier synchronization collecting data.  It is
           assumed that b is either last_pout or last_pout | D
           depending on whether we want to send a 0 or 1 bit....
           Note that the return is the raw (unmapped) port input.
        */
        register portdata x;

        while (P_IN() & GO);      /* wait for barrier reset */
        P_OUT(b | S);             /* synchronize & send data bit */
        while (!(P_IN() & GO)); /* wait for barrier GO signal */
        x = P_IN();               /* get data bit vector (I2,I1,I0) */
        P_OUT(b);                 /* reset synchronization */
        return(x);                /* return data bit vector */
}

extern inline barrier
p_waitvec(register int flag)
{
        /* Do a barrier wait sending flag and return the collected flag
           vector.  This is not super efficient, but is an easier
           interface than using the raw P_OUT() and P_IN() calls
           directly.  Notice that the flag vector bit positions that
           correspond to PEs not in the current barrier are 0.
        */
        register barrier mask = (flag ? OUR_MASK : 0);

        /* If we're not the only PE in our mask, we have work to
           do...  but if we are alone, we are done.
        */
        if (last_mask & ~OUR_MASK) {
                /* Must gather a bit from each PE... */
                register barrier b = (flag ? (last_pout | D) : last_pout);
                register portdata x;
```

```
            /* Translate {I2,I1,I0} into a standard mask */
            mask |= (IToMask(_p_waitvec(b)) & last_mask);
    }

    /* Return constructed bit mask... */
    return(mask);
}
```

5. Conclusion

In this paper, we have presented the complete design of PAPERS, a very simple hardware adapter to allow multiple personal computers to act as one or more fine-grain parallel computers. To support parallel processing, the PAPERS hardware provides full dynamic barrier synchronization, simultaneous broadcast of one data bit from each processor to every other processor, and a variety of maskable interrupt capabilities.

Perhaps the best thing about PAPERS is that it does not require any special interface to the processors; it is connected to ordinary personal computers (or workstations) by their standard Centronics parallel printer ports. However, this also is the worst thing about PAPERS, because these ports often limit performance. Basically, the problem is that many ports are deliberately designed to insert enough wait states so that very slow printer interfaces will be able to sample the output signal without the need for a software delay loop. The speeds of basic operations on PAPERS, assuming a 10' cable, are:

Operation	PAPERS Speed	Speed Using Slow Port
Dynamic barrier sync. with arbitrary PEs	0.2μs	14.0μs
Multiple broadcast communication op.	0.4μs/byte	28.5μs/byte
`ANY` conditional	0.2μs	14.0μs
Random communication involving all PEs	1.6μs/byte	114.0μs/byte

As the table shows, if your machine doesn't have a slow port, PAPERS is a dream come true.... However, if you have a slow port, you might want to either (1) get a better parallel port or (2) wait for us to build a version of PAPERS that directly interfaces to the PC bus.

References

[CoD94] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, "Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors; Part I: Barrier Architecture," submitted to 1994 Int'l Conf. on Parallel Processing.

[CoD94a] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, "Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors; Part II: Mode Emulation," submitted to 1994 Int'l Conf. on Parallel Processing.

[DiO92] H. G. Dietz, M.T. O'Keefe, and A. Zaafrani, "Static Scheduling for Barrier MIMD Architectures," *The Journal of Supercomputing*, vol. 5, pp. 263-289, 1992.

[Jor78] H. F. Jordon, "A Special Purpose Architecture for Finite Element Analysis," *Proc. Int'l Conf. on Parallel Processing*, pp. 263-266, 1978.

[OKD90] 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.

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

[SiN87] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, "The Implementation of the PASM Prototype Control Hierarchy," *Proc. of Second Int'l Conf. on Supercomputing,* pp. I 418-427, 1987.

Table of Contents