Linux Parallel Processing HOWTO

Table of Contents

Linux Parallel Processing HOWTO.....1
Hank Dietz, pplinux@ecn.purdue.edu.....1
1.Introduction.....1
2.SMP Linux.....1
3.Clusters Of Linux Systems.....1
4.SIMD Within A Register (e.g., using MMX).....2
5.Linux−Hosted Attached Processors.....2
6.Of General Interest.....2
1.Introduction.....2
1.1 Is Parallel Processing What I Want?.....2
1.2 Terminology.....3
1.3 Example Algorithm.....7
1.4 Organization Of This Document.....8
2.SMP Linux.....9
2.1 SMP Hardware.....9
Does each processor have its own L2 cache?.....10
Bus configuration?.....10
Memory interleaving and DRAM technologies?.....11
2.2 Introduction To Shared Memory Programming.....11
Shared Everything Vs. Shared Something.....12
Shared Everything.....12
Shared Something.....13
Atomicity And Ordering.....14
Volatility.....14
Locks.....15
Cache Line Size.....16
Linux Scheduler Issues.....16
2.3 bb_threads.....17
2.4 LinuxThreads.....19
2.5 System V Shared Memory.....21
2.6 Memory Map Call.....23
3.Clusters Of Linux Systems.....24
3.1 Why A Cluster?.....24
3.2 Network Hardware.....25
ArcNet.....27
ATM.....27
CAPERS.....28
Ethernet.....28
Ethernet (Fast Ethernet).....29
Ethernet (Gigabit Ethernet).....30
FC (Fibre Channel).....30
FireWire (IEEE 1394).....31
HiPPI And Serial HiPPI.....31
IrDA (Infrared Data Association).....32
Myrinet.....32
Parastation.....33
PLIP.....33

Table of Contents

SCI.....33
SCSI.....34
ServerNet.....34
SHRIMP.....35
SLIP.....35
TTL_PAPERS.....36
USB (Universal Serial Bus).....37
WAPERS.....37
3.3 Network Software Interface.....38
Sockets.....38
UDP Protocol (SOCK_DGRAM).....38
TCP Protocol (SOCK_STREAM).....38
Device Drivers.....39
User−Level Libraries.....39
3.4 PVM (Parallel Virtual Machine).....41
3.5 MPI (Message Passing Interface).....42
3.6 AFAPI (Aggregate Function API).....46
3.7 Other Cluster Support Libraries.....47
Condor (process migration support).....47
DFN−RPC (German Research Network − Remote Procedure Call).....47
DQS (Distributed Queueing System).....48
3.8 General Cluster References.....48
Beowulf.....48
Linux/AP+.....48
Locust.....49
Midway DSM (Distributed Shared Memory).....49
Mosix.....49
NOW (Network Of Workstations).....49
Parallel Processing Using Linux.....49
Pentium Pro Cluster Workshop.....50
TreadMarks DSM (Distributed Shared Memory).....50
U−Net (User−level NETwork interface architecture).....50
WWT (Wisconsin Wind Tunnel).....50
4.SIMD Within A Register (e.g., using MMX).....51
4.1 SWAR: What Is It Good For?.....51
4.2 Introduction To SWAR Programming.....52
Polymorphic Operations.....52
Partitioned Operations.....53
Partitioned Instructions.....53
Unpartitioned Operations With Correction Code.....54
Controlling Field Values.....54
Communication & Type Conversion Operations.....56
Recurrence Operations (Reductions, Scans, etc.).....56
4.3 MMX SWAR Under Linux.....57
5.Linux−Hosted Attached Processors.....59
5.1 A Linux PC Is A Good Host.....59
5.2 Did You DSP That?.....60

Table of Contents

5.3 FPGAs And Reconfigurable Logic Computing.....61
6.Of General Interest.....61
6.1 Programming Languages And Compilers.....62
Fortran 66/77/PCF/90/HPF/95.....62
GLU (Granular Lucid).....63
Jade And SAM.....63
Mentat And Legion.....63
MPL (MasPar Programming Language).....64
PAMS (Parallel Application Management System).....64
Parallaxis−III.....64
pC++/Sage++.....64
SR (Synchronizing Resources).....64
ZPL And IronMan.....65
6.2 Performance Issues.....65
6.3 Conclusion − It's Out There.....66

Linux Parallel Processing HOWTO

Hank Dietz, `pplinux@ecn.purdue.edu`

v980105, 5 January 1998

Parallel Processing *refers to the concept of speeding−up the execution of a program by dividing the program into multiple fragments that can execute simultaneously, each on its own processor. A program being executed across N processors might execute N times faster than it would using a single processor. This document discusses the four basic approaches to parallel processing that are available to Linux users: SMP Linux systems, clusters of networked Linux systems, parallel execution using multimedia instructions (i.e., MMX), and attached (parallel) processors hosted by a Linux system.*

1.Introduction

- 1.1 Is Parallel Processing What I Want?
- 1.2 Terminology
- 1.3 Example Algorithm
- 1.4 Organization Of This Document

2.SMP Linux

- 2.1 SMP Hardware
- 2.2 Introduction To Shared Memory Programming
- 2.3 bb_threads
- 2.4 LinuxThreads
- 2.5 System V Shared Memory
- 2.6 Memory Map Call

3.Clusters Of Linux Systems

- 3.1 Why A Cluster?
- 3.2 Network Hardware
- 3.3 Network Software Interface
- 3.4 PVM (Parallel Virtual Machine)
- 3.5 MPI (Message Passing Interface)
- 3.6 AFAPI (Aggregate Function API)
- 3.7 Other Cluster Support Libraries
- 3.8 General Cluster References

4.SIMD Within A Register (e.g., using MMX)

- 4.1 SWAR: What Is It Good For?
- 4.2 Introduction To SWAR Programming
- 4.3 MMX SWAR Under Linux

5.Linux−Hosted Attached Processors

- 5.1 A Linux PC Is A Good Host
- 5.2 Did You DSP That?
- 5.3 FPGAs And Reconfigurable Logic Computing

6.Of General Interest

- 6.1 Programming Languages And Compilers
- 6.2 Performance Issues
- 6.3 Conclusion − It's Out There

1.Introduction

Parallel Processing refers to the concept of speeding−up the execution of a program by dividing the program into multiple fragments that can execute simultaneously, each on its own processor. A program being executed across *n* processors might execute *n* times faster than it would using a single processor.

Traditionally, multiple processors were provided within a specially designed "parallel computer"; along these lines, Linux now supports **SMP** systems (often sold as "servers") in which multiple processors share a single memory and bus interface within a single computer. It is also possible for a group of computers (for example, a group of PCs each running Linux) to be interconnected by a network to form a parallel−processing **cluster**. The third alternative for parallel computing using Linux is to use the **multimedia instruction extensions** (i.e., MMX) to operate in parallel on vectors of integer data. Finally, it is also possible to use a Linux system as a "host" for a specialized **attached** parallel processing compute engine. All these approaches are discussed in detail in this document.

1.1 Is Parallel Processing What I Want?

Although use of multiple processors can speed−up many operations, most applications cannot yet benefit from parallel processing. Basically, parallel processing is appropriate only if:

- Your application has enough parallelism to make good use of multiple processors. In part, this is a matter of identifying portions of the program that can execute independently and simultaneously on separate processors, but you will also find that some things that *could* execute in parallel might actually slow execution if executed in parallel using a particular system. For example, a program that takes four seconds to execute within a single machine might be able to execute in only one second of processor time on each of four machines, but no speedup would be achieved if it took three seconds or more for these machines to coordinate their actions.
- Either the particular application program you are interested in already has been **parallelized** (rewritten to take advantage of parallel processing) or you are willing to do at least some new coding to take advantage of parallel processing.
- You are interested in researching, or at least becoming familiar with, issues involving parallel processing. Parallel processing using Linux systems isn't necessarily difficult, but it is not familiar to most computer users, and there isn't any book called "Parallel Processing for Dummies"... at least not yet. This HOWTO is a good starting point, not all you need to know.

The good news is that if all the above are true, you'll find that parallel processing using Linux can yield supercomputer performance for some programs that perform complex computations or operate on large data sets. What's more, it can do that using cheap hardware... which you might already own. As an added bonus, it is also easy to use a parallel Linux system for other things when it is not busy executing a parallel job.

If parallel processing is *not* what you want, but you would like to achieve at least a modest improvement in performance, there are still things you can do. For example, you can improve performance of sequential programs by moving to a faster processor, adding memory, replacing an IDE disk with fast wide SCSI, etc. If that's all you are interested in, jump to section 6.2; otherwise, read on.

1.2 Terminology

Although parallel processing has been used for many years in many systems, it is still somewhat unfamiliar to most computer users. Thus, before discussing the various alternatives, it is important to become familiar with a few commonly used terms.

SIMD:

SIMD (Single Instruction stream, Multiple Data stream) refers to a parallel execution model in which all processors execute the same operation at the same time, but each processor is allowed to operate upon its own data. This model naturally fits the concept of performing the same operation on every element of an array, and is thus often associated with vector or array manipulation. Because all operations are inherently synchronized, interactions among SIMD processors tend to be easily and efficiently implemented.

MIMD:

MIMD (Multiple Instruction stream, Multiple Data stream) refers to a parallel execution model in which each processor is essentially acting independently. This model most naturally fits the concept of decomposing a program for parallel execution on a functional basis; for example, one processor might update a database file while another processor generates a graphic display of the new entry. This is a more flexible model than SIMD execution, but it is achieved at the risk of debugging nightmares called **race conditions**, in which a program may intermittently fail due to timing variations reordering the operations of one processor relative to those of another.

SPMD:

SPMD (Single Program, Multiple Data) is a restricted version of MIMD in which all processors are running the same program. Unlike SIMD, each processor executing SPMD code may take a different control flow path through the program.

Communication Bandwidth:

The bandwidth of a communication system is the maximum amount of data that can be transmitted in a unit of time... once data transmission has begun. Bandwidth for serial connections is often measured in **baud** or **bits/second (b/s)**, which generally correspond to 1/10 to 1/8 that many **Bytes/second (B/s)**. For example, a 1,200 baud modem transfers about 120 B/s, whereas a 155 Mb/s ATM network connection is nearly 130,000 times faster, transferring about about 17 MB/s. High bandwidth allows large blocks of data to be transferred efficiently between processors.

Communication Latency:

The latency of a communication system is the minimum time taken to transmit one object, including any send and receive software overhead. Latency is very important in parallel processing because it determines the minimum useful **grain size**, the minimum run time for a segment of code to yield speed−up through parallel execution. Basically, if a segment of code runs for less time than it takes to transmit its result value (i.e., latency), executing that code segment serially on the processor that needed the result value would be faster than parallel execution; serial execution would avoid the communication overhead.

Message Passing:

Message passing is a model for interactions between processors within a parallel system. In general, a message is constructed by software on one processor and is sent through an interconnection network to another processor, which then must accept and act upon the message contents. Although the overhead in handling each message (latency) may be high, there are typically few restrictions on how much information each message may contain. Thus, message passing can yield high bandwidth making it a very effective way to transmit a large block of data from one processor to another. However, to minimize the need for expensive message passing operations, data structures within a parallel program must be spread across the processors so that most data referenced by each processor is in its local memory... this task is known as **data layout**.

Shared Memory:

Shared memory is a model for interactions between processors within a parallel system. Systems like the multi−processor Pentium machines running Linux **physically** share a single memory among their processors, so that a value written to shared memory by one processor can be directly accessed by any processor. Alternatively, **logically** shared memory can be implemented for systems in which each processor has it own memory by converting each non−local memory reference into an appropriate inter−processor communication. Either implementation of shared memory is generally considered easier to use than message passing. Physically shared memory can have both high bandwidth and low latency, but only when multiple processors do not try to access the bus simultaneously; thus, data layout still can seriously impact performance, and cache effects, etc., can make it difficult to determine what the best layout is.

Aggregate Functions:

In both the message passing and shared memory models, a communication is initiated by a single processor; in contrast, aggregate function communication is an inherently parallel communication model in which an entire group of processors act together. The simplest such action is a **barrier synchronization**, in which each individual processor waits until every processor in the group has arrived at the barrier. By having each processor output a datum as a side−effect of reaching a barrier, it is possible to have the communication hardware return a value to each processor which is an arbitrary function of the values collected from all processors. For example, the return value might be the answer to the question "did any processor find a solution?" or it might be the sum of one value from each processor. Latency can be very low, but bandwidth per processor also tends to be low. Traditionally, this model is used primarily to control parallel execution rather than to distribute data values.

Collective Communication:

This is another name for aggregate functions, most often used when referring to aggregate functions that are constructed using multiple message−passing operations.

SMP:

SMP (Symmetric Multi−Processor) refers to the operating system concept of a group of processors working together as peers, so that any piece of work could be done equally well by any processor. Typically, SMP implies the combination of MIMD and shared memory. In the IA32 world, SMP generally means compliant with MPS (the Intel MultiProcessor Specification); in the future, it may mean "Slot 2"....

SWAR:

SWAR (SIMD Within A Register) is a generic term for the concept of partitioning a register into multiple integer fields and using register−width operations to perform SIMD−parallel computations across those fields. Given a machine with *k*−bit registers, data paths, and function units, it has long been known that ordinary register operations can function as SIMD parallel operations on as many as *n*, *k*/*n*−bit, field values. Although this type of parallelism can be implemented using ordinary integer registers and instructions, many high−end microprocessors have recently added specialized instructions to enhance the performance of this technique for multimedia−oriented tasks. In addition to the Intel/AMD/Cyrix **MMX** (MultiMedia eXtensions), there are: Digital Alpha **MAX** (MultimediA eXtensions), Hewlett−Packard PA−RISC **MAX** (Multimedia Acceleration eXtensions), MIPS **MDMX** (Digital Media eXtension, pronounced "Mad Max"), and Sun SPARC V9 **VIS** (Visual Instruction Set). Aside from the three vendors who have agreed on MMX, all of these instruction set extensions are roughly comparable, but mutually incompatible.

Attached Processors:

Attached processors are essentially special−purpose computers that are connected to a **host** system to accelerate specific types of computation. For example, many video and audio cards for PCs contain attached processors designed, respectively, to accelerate common graphics operations and audio **DSP** (Digital Signal Processing). There is also a wide range of attached **array processors**, so called because they are designed to accelerate arithmetic operations on arrays. In fact, many commercial supercomputers are really attached processors with workstation hosts.

RAID:

RAID (Redundant Array of Inexpensive Disks) is a simple technology for increasing both the bandwidth and reliability of disk I/O. Although there are many different variations, all have two key concepts in common. First, each data block is **striped** across a group of *n+k* disk drives such that each drive only has to read or write 1/*n* of the data... yielding *n* times the bandwidth of one drive. Second, redundant data is written so that data can be recovered if a disk drive fails; this is important because otherwise if any one of the *n+k* drives were to fail, the entire file system could be lost. A good overview of RAID in general is given at http://www.dpt.com/uraiddoc.html, and information about RAID options for Linux systems is at http://linas.org/linux/raid.html. Aside from specialized RAID hardware support, Linux also supports software RAID 0, 1, 4, and 5 across multiple disks hosted by a single Linux system; see the Software RAID mini−HOWTO and the Multi−Disk System Tuning mini−HOWTO for details. RAID across disk drives *on multiple machines in a cluster* is not directly supported.

IA32:

IA32 (Intel Architecture, 32−bit) really has nothing to do with parallel processing, but rather refers to the class of processors whose instruction sets are generally compatible with that of the Intel 386. Basically, any Intel x86 processor after the 286 is compatible with the 32−bit flat memory model that characterizes IA32. AMD and Cyrix also make a multitude of IA32−compatible processors. Because Linux evolved primarily on IA32 processors and that is where the commodity market is centered, it is convenient to use IA32 to distinguish any of these processors from the PowerPC, Alpha, PA−RISC, MIPS, SPARC, etc. The upcoming IA64 (64−bit with EPIC, Explicitly Parallel Instruction Computing) will certainly complicate matters, but Merced, the first IA64 processor, is not scheduled for production until 1999.

COTS:

Since the demise of many parallel supercomputer companies, COTS (Commercial Off−The−Shelf) is commonly discussed as a requirement for parallel computing systems. Being fanatically pure, the only COTS parallel processing techniques using PCs are things like SMP Windows NT servers and various MMX Windows applications; it really doesn't pay to be that fanatical. The underlying concept of COTS is really minimization of development time and cost. Thus, a more useful, more common, meaning of COTS is that at least most subsystems benefit from commodity marketing, but other technologies are used where they are effective. Most often, COTS parallel processing refers to a cluster in which the nodes are commodity PCs, but the network interface and software are somewhat customized... typically running Linux and applications codes that are freely available (e.g., copyleft or public domain), but not literally COTS.

1.3 Example Algorithm

In order to better understand the use of the various parallel programming approaches outlined in this HOWTO, it is useful to have an example problem. Although just about any simple parallel algorithm would do, by selecting an algorithm that has been used to demonstrate various other parallel programming systems, it becomes a bit easier to compare and contrast approaches. M. J. Quinn's book, *Parallel Computing Theory And Practice*, second edition, McGraw Hill, New York, 1994, uses a parallel algorithm that computes the value of Pi to demonstrate a variety of different parallel supercomputer programming environments (e.g., nCUBE message passing, Sequent shared memory). In this HOWTO, we use the same basic algorithm.

The algorithm computes the approximate value of Pi by summing the area under *x* squared. As a purely sequential C program, the algorithm looks like:

```
#include <stdlib.h>;
#include <stdio.h>;

main(int argc, char **argv)
{
  register double width, sum;
  register int intervals, i;

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  /* do the computation */
  sum = 0;
  for (i=0; i<intervals; ++i) {
    register double x = (i + 0.5) * width;
    sum += 4.0 / (1.0 + x * x);
  }
  sum *= width;

  printf("Estimation of pi is %f\n", sum);

  return(0);
}
```

However, this sequential algorithm easily yields an "embarrassingly parallel" implementation. The area is subdivided into intervals, and any number of processors can each independently sum the intervals assigned to it, with no need for interaction between processors. Once the local sums have been computed, they are added together to create a global sum; this step requires some level of coordination and communication between processors. Finally, this global sum is printed by one processor as the approximate value of Pi.

In this HOWTO, the various parallel implementations of this algorithm appear where each of the different programming methods is discussed.

1.4 Organization Of This Document

The remainder of this document is divided into five parts. Sections 2, 3, 4, and 5 correspond to the three different types of hardware configurations supporting parallel processing using Linux:

- Section 2 discusses SMP Linux systems. These directly support MIMD execution using shared memory, although message passing also is implemented easily. Although Linux supports SMP configurations up to 16 processors, most SMP PC systems have either two or four identical processors.
- Section 3 discusses clusters of networked machines, each running Linux. A cluster can be used as a parallel processing system that directly supports MIMD execution and message passing, perhaps also providing logically shared memory. Simulated SIMD execution and aggregate function communication also can be supported, depending on the networking method used. The number of processors in a cluster can range from two to thousands, primarily limited by the physical wiring constraints of the network. In some cases, various types of machines can be mixed within a cluster; for example, a network combining DEC Alpha and Pentium Linux systems would be a **heterogeneous cluster**.
- Section 4 discusses SWAR, SIMD Within A Register. This is a very restrictive type of parallel execution model, but on the other hand, it is a built−in capability of ordinary processors. Recently, MMX (and other) instruction set extensions to modern processors have made this approach even more effective.
- Section 5 discusses the use of Linux PCs as hosts for simple parallel computing systems. Either as an add−in card or as an external box, attached processors can provide a Linux system with formidable processing power for specific types of applications. For example, inexpensive ISA cards are available that provide multiple DSP processors offering hundreds of MFLOPS for compute−bound problems. However, these add−in boards are *just* processors; they generally do not run an OS, have disk or console I/O capability, etc. To make such systems useful, the Linux "host" must provide these functions.

The final section of this document covers aspects that are of general interest for parallel processing using Linux, not specific to a particular one of the approaches listed above.

As you read this document, keep in mind that we haven't tested everything, and a lot of stuff reported here "still has a research character" (a nice way to say "doesn't quite work like it should" ;−). However, parallel processing using Linux is useful now, and an increasingly large group is working to make it better.

The author of this HOWTO is Hank Dietz, Ph.D., currently Associate Professor of Electrical and Computer Engineering at Purdue University, in West Lafayette, IN, 47907−1285. Dietz retains rights to this document as per the Linux Documentation Project guidelines. Although an effort has been made to ensure the correctness and fairness of this presentation, neither Dietz nor Purdue University can be held responsible for any problems or errors, and Purdue University does not endorse any of the work/products discussed.

2.SMP Linux

This document gives a brief overview of how to use SMP Linux systems for parallel processing. The most up−to−date information on SMP Linux is probably available via the SMP Linux project mailing list; send email to majordomo@vger.rutgers.edu with the text `subscribe linux−smp` to join the list.

Does SMP Linux really work? In June 1996, I purchased a brand new (well, new off−brand ;−) two−processor 100MHz Pentium system. The fully assembled system, including both processors, Asus motherboard, 256K cache, 32M RAM, 1.6G disk, 6X CDROM, Stealth 64, and 15" Acer monitor, cost a total of $1,800. This was just a few hundred dollars more than a comparable uniprocessor system. Getting SMP Linux running was simply a matter of installing the "stock" uniprocessor Linux, recompiling the kernel with the `SMP=1` line in the makefile uncommented (although I find setting `SMP` to `1` a bit ironic ;−), and informing `lilo` about the new kernel. This system performs well enough, and has been stable enough, to serve as my primary workstation ever since. In summary, SMP Linux really does work.

The next question is how much high−level support is available for writing and executing shared memory parallel programs under SMP Linux. Through early 1996, there wasn't much. Things have changed. For example, there is now a very complete POSIX threads library.

Although performance may be lower than for native shared−memory mechanisms, an SMP Linux system also can use most parallel processing software that was originally developed for a workstation cluster using socket communication. Sockets (see section 3.3) work within an SMP Linux system, and even for multiple SMPs networked as a cluster. However, sockets imply a lot of unnecessary overhead for an SMP. Much of that overhead is within the kernel or interrupt handlers; this worsens the problem because SMP Linux generally allows only one processor to be in the kernel at a time and the interrupt controller is set so that only the boot processor can process interrupts. Despite this, typical SMP communication hardware is so much better than most cluster networks that cluster software will often run better on an SMP than on the cluster for which it was designed.

The remainder of this section discusses SMP hardware, reviews the basic Linux mechanisms for sharing memory across the processes of a parallel program, makes a few observations about atomicity, volatility, locks, and cache lines, and finally gives some pointers to other shared memory parallel processing resources.

2.1 SMP Hardware

Although SMP systems have been around for many years, until very recently, each such machine tended to implement basic functions differently enough so that operating system support was not portable. The thing that has changed this situation is Intel's Multiprocessor Specification, often referred to as simply **MPS**. The MPS 1.4 specification is currently available as a PDF file at http://www.intel.com/design/pro/datashts/242016.htm, and there is a brief overview of MPS 1.1 at http://support.intel.com/oem_developer/ial/support/9300.HTM, but be aware that Intel does re−arrange their WWW site often. A wide range of vendors are building MPS−compliant systems supporting up to four processors, but MPS theoretically allows many more processors.

The only non−MPS, non−IA32, systems supported by SMP Linux are Sun4m multiprocessor SPARC machines. SMP Linux supports most Intel MPS version 1.1 or 1.4 compliant machines with up to sixteen

486DX, Pentium, Pentium MMX, Pentium Pro, or Pentium II processors. Unsupported IA32 processors include the Intel 386, Intel 486SX/SLC processors (the lack of floating point hardware interferes with the SMP mechanisms), and AMD & Cyrix processors (they require different SMP support chips that do not seem to be available at this writing).

It is important to understand that the performance of MPS−compliant systems can vary widely. As expected, one cause for performance differences is processor speed: faster clock speeds tend to yield faster systems, and a Pentium Pro processor is faster than a Pentium. However, MPS does not really specify how hardware implements shared memory, but only how that implementation must function from a software point of view; this means that performance is also a function of how the shared memory implementation interacts with the characteristics of SMP Linux and your particular programs.

The primary way in which systems that comply with MPS differ is in how they implement access to physically shared memory.

Does each processor have its own L2 cache?

Some MPS Pentium systems, and all MPS Pentium Pro and Pentium II systems, have independent L2 caches. (The L2 cache is packaged within the Pentium Pro or Pentium II modules.) Separate L2 caches are generally viewed as maximizing compute performance, but things are not quite so obvious under Linux. The primary complication is that the current SMP Linux scheduler does not attempt to keep each process on the same processor, a concept known as **processor affinity**. This may change soon; there has recently been some discussion about this in the SMP Linux development community under the title "processor binding." Without processor affinity, having separate L2 caches may introduce significant overhead when a process is given a timeslice on a processor other than the one that was executing it last.

Many relatively inexpensive systems are organized so that two Pentium processors share a single L2 cache. The bad news is that this causes contention for the cache, seriously degrading performance when running multiple independent sequential programs. The good news is that many parallel programs might actually benefit from the shared cache because if both processors will want to access the same line from shared memory, only one had to fetch it into cache and contention for the bus is averted. The lack of processor affinity also causes less damage with a shared L2 cache. Thus, for parallel programs, it isn't really clear that sharing L2 cache is as harmful as one might expect.

Experience with our dual Pentium shared 256K cache system shows quite a wide range of performance depending on the level of kernel activity required. At worst, we see only about 1.2x speedup. However, we also have seen up to 2.1x speedup, which suggests that compute−intensive SPMD−style code really does profit from the "shared fetch" effect.

Bus configuration?

The first thing to say is that most modern systems connect the processors to one or more PCI buses that in turn are "bridged" to one or more ISA/EISA buses. These bridges add latency, and both EISA and ISA generally offer lower bandwidth than PCI (ISA being the lowest), so disk drives, video cards, and other high−performance devices generally should be connected via a PCI bus interface.

Although an MPS system can achieve good speed−up for many compute−intensive parallel programs even if there is only one PCI bus, I/O operations occur at no better than uniprocessor performance... and probably a little worse due to bus contention from the processors. Thus, if you are looking to speed−up I/O, make sure that you get an MPS system with multiple independent PCI busses and I/O controllers (e.g., multiple SCSI chains). You will need to be careful to make sure SMP Linux supports what you get. Also keep in mind that the current SMP Linux essentially allows only one processor in the kernel at any time, so you should choose your I/O controllers carefully to pick ones that minimize the kernel time required for each I/O operation. For really high performance, you might even consider doing raw device I/O directly from user processes, without a system call... this isn't necessarily as hard as it sounds, and need not compromise security (see section 3.3 for a description of the basic techniques).

It is important to note that the relationship between bus speed and processor clock rate has become very fuzzy over the past few years. Although most systems now use the same PCI clock rate, it is not uncommon to find a faster processor clock paired with a slower bus clock. The classic example of this was that the Pentium 133 generally used a faster bus than a Pentium 150, with appropriately strange−looking performance on various benchmarks. These effects are amplified in SMP systems; it is even more important to have a faster bus clock.

Memory interleaving and DRAM technologies?

Memory interleaving actually has nothing whatsoever to do with MPS, but you will often see it mentioned for MPS systems because these systems are typically more demanding of memory bandwidth. Basically, two−way or four−way interleaving organizes RAM so that a block access is accomplished using multiple banks of RAM rather than just one. This provides higher memory access bandwidth, particularly for cache line loads and stores.

The waters are a bit muddied about this, however, because EDO DRAM and various other memory technologies tend to improve similar kinds of operations. An excellent overview of DRAM technologies is given in http://www.pcguide.com/ref/ram/tech.htm.

So, for example, is it better to have 2−way interleaved EDO DRAM or non−interleaved SDRAM? That is a very good question with no simple answer, because both interleaving and exotic DRAM technologies tend to be expensive. The same dollar investment in more ordinary memory configurations generally will give you a significantly larger main memory. Even the slowest DRAM is still a heck of a lot faster than using disk−based virtual memory....

2.2 Introduction To Shared Memory Programming

Ok, so you have decided that parallel processing on an SMP is a great thing to do... how do you get started? Well, the first step is to learn a little bit about how shared memory communication really works.

It sounds like you simply have one processor store a value into memory and another processor load it; unfortunately, it isn't quite that simple. For example, the relationship between processes and processors is very blurry; however, if we have no more active processes than there are processors, the terms are roughly interchangeable. The remainder of this section briefly summarizes the key issues that could cause serious problems, if you were not aware of them: the two different models used to determine what is shared, atomicity issues, the concept of volatility, hardware lock instructions, cache line effects, and Linux scheduler issues.

Shared Everything Vs. Shared Something

There are two fundamentally different models commonly used for shared memory programming: **shared everything** and **shared something**. Both of these models allow processors to communicate by loads and stores from/into shared memory; the distinction comes in the fact that shared everything places all data structures in shared memory, while shared something requires the user to explicitly indicate which data structures are potentially shared and which are **private** to a single processor.

Which shared memory model should you use? That is mostly a question of religion. A lot of people like the shared everything model because they do not really need to identify which data structures should be shared at the time they are declared... you simply put locks around potentially−conflicting accesses to shared objects to ensure that only one process(or) has access at any moment. Then again, that really isn't all that simple... so many people prefer the relative safety of shared something.

Shared Everything

The nice thing about sharing everything is that you can easily take an existing sequential program and incrementally convert it into a shared everything parallel program. You do not have to first determine which data need to be accessible by other processors.

Put simply, the primary problem with sharing everything is that any action taken by one processor could affect the other processors. This problem surfaces in two ways:

- Many libraries use data structures that simply are not sharable. For example, the UNIX convention is that most functions can return an error code in a variable called `errno`; if two shared everything processes perform various calls, they would interfere with each other because they share the same `errno`. Although there is now a library version that fixes the `errno` problem, similar problems still exist in most libraries. For example, unless special precautions are taken, the X library will not work if calls are made from multiple shared everything processes.
- Normally, the worst−case behavior for a program with a bad pointer or array subscript is that the process that contains the offending code dies. It might even generate a `core` file that clues you in to what happened. In shared everything parallel processing, it is very likely that the stray accesses will bring the demise of *a process other than the one at fault*, making it nearly impossible to localize and correct the error.

Neither of these types of problems is common when shared something is used, because only the explicitly−marked data structures are shared. It also is fairly obvious that shared everything only works if all processors are executing the exact same memory image; you cannot use shared everything across multiple different code images (i.e., can use only SPMD, not general MIMD).

The most common type of shared everything programming support is a **threads library**. Threads are essentially "light−weight" processes that might not be scheduled in the same way as regular UNIX processes and, most importantly, share access to a single memory map. The POSIX Pthreads package has been the focus of a number of porting efforts; the big question is whether any of these ports actually run the threads of a program in parallel under SMP Linux (ideally, with a processor for each thread). The POSIX API doesn't require it, and versions like http://www.aa.net/~mtp/PCthreads.html apparently do not implement parallel thread execution − all the threads of a program are kept within a single Linux process.

The first threads library that supported SMP Linux parallelism was the now somewhat obsolete bb_threads library, ftp://caliban.physics.utoronto.ca/pub/linux/, a very small library that used the Linux `clone()` call to fork new, independently scheduled, Linux processes all sharing a single address space. SMP Linux machines can run multiple of these "threads" in parallel because each "thread" is a full Linux process; the trade−off is that you do not get the same "light−weight" scheduling control provided by some thread libraries under other operating systems. The library used a bit of C−wrapped assembly code to install a new chunk of memory as each thread's stack and to provide atomic access functions for an array of locks (mutex objects). Documentation consisted of a `README` and a short sample program.

More recently, a version of POSIX threads using `clone()` has been developed. This library, LinuxThreads, is clearly the preferred shared everything library for use under SMP Linux. POSIX threads are well documented, and the LinuxThreads README and LinuxThreads FAQ are very well done. The primary problem now is simply that POSIX threads have a lot of details to get right and LinuxThreads is still a work in progress. There is also the problem that the POSIX thread standard has evolved through the standardization process, so you need to be a bit careful not to program for obsolete early versions of the standard.

Shared Something

Shared something is really "only share what needs to be shared." This approach can work for general MIMD (not just SPMD) provided that care is taken for the shared objects to be allocated at the same places in each processor's memory map. More importantly, shared something makes it easier to predict and tune performance, debug code, etc. The only problems are:

- It can be hard to know beforehand what really needs to be shared.
- The actual allocation of objects in shared memory may be awkward, especially for what would have been stack−allocated objects. For example, it may be necessary to explicitly allocate shared objects in a separate memory segment, requiring separate memory allocation routines and introducing extra pointer indirections in each reference.

Currently, there are two very similar mechanisms that allow groups of Linux processes to have independent memory spaces, all sharing only a relatively small memory segment. Assuming that you didn't foolishly exclude "System V IPC" when you configured your Linux system, Linux supports a very portable mechanism that has generally become known as "System V Shared Memory." The other alternative is a memory mapping facility whose implementation varies widely across different UNIX systems: the `mmap()` system call. You can, and should, learn about these calls from the manual pages... but a brief overview of each is given in sections 2.5 and 2.6 to help get you started.

Atomicity And Ordering

No matter which of the above two models you use, the result is pretty much the same: you get a pointer to a chunk of read/write memory that is accessible by all processes within your parallel program. Does that mean I can just have my parallel program access shared memory objects as though they were in ordinary local memory? Well, not quite....

Atomicity refers to the concept that an operation on an object is accomplished as an indivisible, uninterruptible, sequence. Unfortunately, sharing memory access does not imply that all operations on data in shared memory occur atomically. Unless special precautions are taken, only simple load or store operations that occur within a single bus transaction (i.e., aligned 8, 16, or 32−bit operations, but not misaligned nor 64−bit operations) are atomic. Worse still, "smart" compilers like GCC will often perform optimizations that could eliminate the memory operations needed to ensure that other processors can see what this processor has done. Fortunately, both these problems can be remedied... leaving only the relationship between access efficiency and cache line size for us to worry about.

However, before discussing these issues, it is useful to point−out that all of this assumes that memory references for each processor happen in the order in which they were coded. The Pentium does this, but also notes that future Intel processors might not. So, for future processors, keep in mind that it may be necessary to surround some shared memory accesses with instructions that cause all pending memory accesses to complete, thus providing memory access ordering. The `CPUID` instruction apparently is reserved to have this side−effect.

Volatility

To prevent GCC's optimizer from buffering values of shared memory objects in registers, all objects in shared memory should be declared as having types with the `volatile` attribute. If this is done, all shared object reads and writes that require just one word access will occur atomically. For example, suppose that *p* is a pointer to an integer, where both the pointer and the integer it will point at are in shared memory; the ANSI C declaration might be:

```
volatile int * volatile p;
```

In this code, the first `volatile` refers to the `int` that `p` will eventually point at; the second `volatile` refers to the pointer itself. Yes, it is annoying, but it is the price one pays for enabling GCC to perform some very powerful optimizations. At least in theory, the `−traditional` option to GCC might suffice to produce correct code at the expense of some optimization, because pre−ANSI K&R C essentially claimed that all variables were volatile unless explicitly declared as `register`. Still, if your typical GCC compile looks like `cc −O6 ...`, you really will want to explicitly mark things as volatile only where necessary.

There has been a rumor to the effect that using assembly−language locks that are marked as modifying all processor registers will cause GCC to appropriately flush all variables, thus avoiding the "inefficient" compiled code associated with things declared as `volatile`. This hack appears to work for statically allocated global variables using version 2.7.0 of GCC... however, that behavior is *not* required by the ANSI C standard. Still worse, other processes that are making only read accesses can buffer the values in registers forever, thus *never* noticing that the shared memory value has actually changed. In summary, do what you want, but only variables accessed through `volatile` are *guaranteed* to work correctly.

Note that you can cause a volatile access to an ordinary variable by using a type cast that imposes the `volatile` attribute. For example, the ordinary `int i;` can be referenced as a volatile by `*((volatile int *) &i)`; thus, you can explicitly invoke the "overhead" of volatility only where it is critical.

Locks

If you thought that `++i;` would always work to add one to a variable `i` in shared memory, you've got a nasty little surprise coming: even if coded as a single instruction, the load and store of the result are separate memory transactions, and other processors could access `i` between these two transactions. For example, having two processes both perform `++i;` might only increment `i` by one, rather than by two. According to the Intel Pentium "Architecture and Programming Manual," the `LOCK` prefix can be used to ensure that any of the following instructions is atomic relative to the data memory location it accesses:

```
BTS, BTR, BTC                     mem, reg/imm
XCHG                              reg, mem
XCHG                              mem, reg
ADD, OR, ADC, SBB, AND, SUB, XOR  mem, reg/imm
NOT, NEG, INC, DEC                mem
CMPXCHG, XADD
```

However, it probably is not a good idea to use all these operations. For example, `XADD` did not even exist for the 386, so coding it may cause portability problems.

The `XCHG` instruction *always* asserts a lock, even without the `LOCK` prefix, and thus is clearly the preferred atomic operation from which to build higher−level atomic constructs such as semaphores and shared queues. Of course, you can't get GCC to generate this instruction just by writing C code... instead, you must use a bit of in−line assembly code. Given a word−size volatile object *obj* and a word−size register value *reg*, the GCC in−line assembly code is:

```
__asm__ __volatile__ ("xchgl %1,%0"
                      :"=r" (reg), "=m" (obj)
                      :"r" (reg), "m" (obj));
```

Examples of GCC in−line assembly code using bit operations for locking are given in the source code for the bb_threads library.

It is important to remember, however, that there is a cost associated with making memory transactions atomic. A locking operation carries a fair amount of overhead and may delay memory activity from other processors, whereas ordinary references may use local cache. The best performance results when locking operations are used as infrequently as possible. Further, these IA32 atomic instructions obviously are not portable to other systems.

There are many alternative approaches that allow ordinary instructions to be used to implement various synchronizations, including **mutual exclusion** − ensuring that at most one processor is updating a given shared object at any moment. Most OS textbooks discuss at least one of these techniques. There is a fairly good discussion in the Fourth Edition of *Operating System Concepts*, by Abraham Silberschatz and Peter B. Galvin, ISBN 0−201−50480−4.

Cache Line Size

One more fundamental atomicity concern can have a dramatic impact on SMP performance: cache line size. Although the MPS standard requires references to be coherent no matter what caching is used, the fact is that when one processor writes to a particular line of memory, every cached copy of the old line must be invalidated or updated. This implies that if two or more processors are both writing data to different portions of the same line a lot of cache and bus traffic may result, effectively to pass the line from cache to cache. This problem is known as **false sharing**. The solution is simply to try to *organize data so that what is accessed in parallel tends to come from a different cache line for each process*.

You might be thinking that false sharing is not a problem using a system with a shared L2 cache, but remember that there are still separate L1 caches. Cache organization and number of separate levels can both vary, but the Pentium L1 cache line size is 32 bytes and typical external cache line sizes are around 256 bytes. Suppose that the addresses (physical or virtual) of two items are *a* and *b* and that the largest per−processor cache line size is *c*, which we assume to be a power of two. To be very precise, if `((int) a) & ~(c - 1)` is equal to `((int) b) & ~(c - 1)`, then both references are in the same cache line. A simpler rule is that if shared objects being referenced in parallel are at least *c* bytes apart, they should map to different cache lines.

Linux Scheduler Issues

Although the whole point of using shared memory for parallel processing is to avoid OS overhead, OS overhead can come from things other than communication per se. We have already said that the number of processes that should be constructed is less than or equal to the number of processors in the machine. But how do you decide exactly how many processes to make?

For best performance, *the number of processes in your parallel program should be equal to the expected number of your program's processes that simultaneously can be running on different processors*. For example, if a four−processor SMP typically has one process actively running for some other purpose (e.g., a WWW server), then your parallel program should use only three processes. You can get a rough idea of how many other processes are active on your system by looking at the "load average" quoted by the `uptime` command.

Alternatively, you could boost the priority of the processes in your parallel program using, for example, the `renice` command or `nice()` system call. You must be privileged to increase priority. The idea is simply to force the other processes out of processors so that your program can run simultaneously across all processors. This can be accomplished somewhat more explicitly using the prototype version of SMP Linux at http://luz.cs.nmt.edu/~rtlinux/, which offers real−time schedulers.

If you are not the only user treating your SMP system as a parallel machine, you may also have conflicts between the two or more parallel programs trying to execute simultaneously. This standard solution is **gang scheduling** − i.e., manipulating scheduling priority so that at any given moment, only the processes of a single parallel program are running. It is useful to recall, however, that using more parallelism tends to have diminishing returns and scheduler activity adds overhead. Thus, for example, it is probably better for a four−processor machine to run two programs with two processes each rather than gang scheduling between two programs with four processes each.

There is one more twist to this. Suppose that you are developing a program on a machine that is heavily used all day, but will be fully available for parallel execution at night. You need to write and test your code for correctness with the full number of processes, even though you know that your daytime test runs will be slow. Well, they will be *very* slow if you have processes **busy waiting** for shared memory values to be changed by other processes that are not currently running (on other processors). The same problem occurs if you develop and test your code on a single−processor system.

The solution is to embed calls in your code, wherever it may loop awaiting an action from another processor, so that Linux will give another process a chance to run. I use a C macro, call it `IDLE_ME`, to do this: for a test run, compile with `cc -DIDLE_ME=usleep(1); ...`; for a "production" run, compile with `cc -DIDLE_ME={} ...`. The `usleep(1)` call requests a 1 microsecond sleep, which has the effect of allowing the Linux scheduler to select a different process to run on that processor. If the number of processes is more than twice the number of processors available, it is not unusual for codes to run ten times faster with `usleep(1)` calls than without them.

2.3 bb_threads

The bb_threads ("Bare Bones" threads) library, ftp://caliban.physics.utoronto.ca/pub/linux/, is a remarkably simple library that demonstrates use of the Linux `clone()` call. The `gzip tar` file is only 7K bytes! Although this library is essentially made obsolete by the LinuxThreads library discussed in section 2.4, bb_threads is still usable, and it is small and simple enough to serve well as an introduction to use of Linux thread support. Certainly, it is far less daunting to read this source code than to browse the source code for LinuxThreads. In summary, the bb_threads library is a good starting point, but is not really suitable for coding large projects.

The basic program structure for using the bb_threads library is:

1. Start the program running as a single process.
2. You will need to estimate the maximum stack space that will be required for each thread. Guessing large is relatively harmless (that is what virtual memory is for ;−), but remember that *all* the stacks are coming from a single virtual address space, so guessing huge is not a great idea. The demo suggests 64K. This size is set to *b* bytes by `bb_threads_stacksize(b)`.
3. The next step is to initialize any locks that you will need. The lock mechanism built−into this library numbers locks from 0 to `MAX_MUTEXES`, and initializes lock *i* by `bb_threads_mutexcreate(i)`.
4. Spawning a new thread is done by calling a library routine that takes arguments specifying what function the new thread should execute and what arguments should be transmitted to it. To start a new thread executing the `void`−returning function *f* with the single argument *arg*, you do something like `bb_threads_newthread(f, &arg)`, where *f* should be declared something like `void f(void *arg, size_t dummy)`. If you need to pass more than one argument, pass a pointer to a structure initialized to hold the argument values.
5. Run parallel code, being careful to use `bb_threads_lock(n)` and

bb_threads_unlock(*n*) where *n* is an integer identifying which lock to use. Note that the lock and unlock operations in this library are very basic spin locks using atomic bus−locking instructions, which can cause excessive memory−reference interference and do not make any attempt to ensure fairness. The demo program packaged with bb_threads did not correctly use locks to prevent `printf()` from being executed simultaneously from within the functions `fnn` and `main`... and because of this, the demo does not always work. I'm not saying this to knock the demo, but rather to emphasize that this stuff is *very tricky*; also, it is only slightly easier using LinuxThreads.

6. When a thread executes a `return`, it actually destroys the process... but the local stack memory is not automatically deallocated. To be precise, Linux doesn't support deallocation, but the memory space is not automatically added back to the `malloc()` free list. Thus, the parent process should reclaim the space for each dead child by `bb_threads_cleanup(wait(NULL))`.

The following C program uses the algorithm discussed in section 1.3 to compute the approximate value of Pi using two bb_threads threads.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "bb_threads.h"

volatile double pi = 0.0;
volatile int intervals;
volatile int pids[2];      /* Unix PIDs of threads */

void
do_pi(void *data, size_t len)
{
  register double width, localsum;
  register int i;
  register int iproc = (getpid() != pids[0]);

  /* set width */
  width = 1.0 / intervals;

  /* do the local computations */
  localsum = 0;
  for (i=iproc; i<intervals; i+=2) {
    register double x = (i + 0.5) * width;
    localsum += 4.0 / (1.0 + x * x);
  }
  localsum *= width;

  /* get permission, update pi, and unlock */
  bb_threads_lock(0);
  pi += localsum;
  bb_threads_unlock(0);
}

int
main(int argc, char **argv)
{
  /* get the number of intervals */
  intervals = atoi(argv[1]);
```

```
  /* set stack size and create lock... */
  bb_threads_stacksize(65536);
  bb_threads_mutexcreate(0);

  /* make two threads... */
  pids[0] = bb_threads_newthread(do_pi, NULL);
  pids[1] = bb_threads_newthread(do_pi, NULL);

  /* cleanup after two threads (really a barrier sync) */
  bb_threads_cleanup(wait(NULL));
  bb_threads_cleanup(wait(NULL));

  /* print the result */
  printf("Estimation of pi is %f\n", pi);

  /* check-out */
  exit(0);
}
```

2.4 LinuxThreads

LinuxThreads http://pauillac.inria.fr/~xleroy/linuxthreads/ is a fairly complete and solid implementation of "shared everything" as per the POSIX 1003.1c threads standard. Unlike other POSIX threads ports, LinuxThreads uses the same Linux kernel threads facility (`clone()`) that is used by bb_threads. POSIX compatibility means that it is relatively easy to port quite a few threaded applications from other systems and various tutorial materials are available. In short, this is definitely the threads package to use under Linux for developing large–scale threaded programs.

The basic program structure for using the LinuxThreads library is:

1. Start the program running as a single process.
2. The next step is to initialize any locks that you will need. Unlike bb_threads locks, which are identified by numbers, POSIX locks are declared as variables of type `pthread_mutex_t lock`. Use `pthread_mutex_init(&lock,val)` to initialize each one you will need to use.
3. As with bb_threads, spawning a new thread is done by calling a library routine that takes arguments specifying what function the new thread should execute and what arguments should be transmitted to it. However, POSIX requires the user to declare a variable of type `pthread_t` to identify each thread. To create a thread `pthread_t thread` running `f()`, one calls `pthread_create(&thread,NULL,f,&arg)`.
4. Run parallel code, being careful to use `pthread_mutex_lock(&lock)` and `pthread_mutex_unlock(&lock)` as appropriate.
5. Use `pthread_join(thread,&retval)` to clean–up after each thread.
6. Use `–D_REENTRANT` when compiling your C code.

An example parallel computation of Pi using LinuxThreads follows. The algorithm of section 1.3 is used and, as for the bb_threads example, two threads execute in parallel.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include "pthread.h"

volatile double pi = 0.0;  /* Approximation to pi (shared) */
pthread_mutex_t pi_lock;   /* Lock for above */
volatile double intervals; /* How many intervals? */

void *
process(void *arg)
{
  register double width, localsum;
  register int i;
  register int iproc = (*((char *) arg) - '0');

  /* Set width */
  width = 1.0 / intervals;

  /* Do the local computations */
  localsum = 0;
  for (i=iproc; i<intervals; i+=2) {
    register double x = (i + 0.5) * width;
    localsum += 4.0 / (1.0 + x * x);
  }
  localsum *= width;

  /* Lock pi for update, update it, and unlock */
  pthread_mutex_lock(&pi_lock);
  pi += localsum;
  pthread_mutex_unlock(&pi_lock);

  return(NULL);
}

int
main(int argc, char **argv)
{
  pthread_t thread0, thread1;
  void * retval;

  /* Get the number of intervals */
  intervals = atoi(argv[1]);

  /* Initialize the lock on pi */
  pthread_mutex_init(&pi_lock, NULL);

  /* Make the two threads */
  if (pthread_create(&thread0, NULL, process, "0") ||
      pthread_create(&thread1, NULL, process, "1")) {
    fprintf(stderr, "%s: cannot make thread\n", argv[0]);
    exit(1);
  }

  /* Join (collapse) the two threads */
  if (pthread_join(thread0, &retval) ||
      pthread_join(thread1, &retval)) {
    fprintf(stderr, "%s: thread join failed\n", argv[0]);
    exit(1);
  }

  /* Print the result */
  printf("Estimation of pi is %f\n", pi);

  /* Check-out */
```

```
  exit(0);
}
```

2.5 System V Shared Memory

The System V IPC (Inter−Process Communication) support consists of a number of system calls providing message queues, semaphores, and a shared memory mechanism. Of course, these mechanisms were originally intended to be used for multiple processes to communicate within a uniprocessor system. However, that implies that it also should work to communicate between processes under SMP Linux, no matter which processors they run on.

Before going into how these calls are used, it is important to understand that although System V IPC calls exist for things like semaphores and message transmission, you probably should not use them. Why not? These functions are generally slow and serialized under SMP Linux. Enough said.

The basic procedure for creating a group of processes sharing access to a shared memory segment is:

1. Start the program running as a single process.
2. Typically, you will want each run of a parallel program to have its own shared memory segment, so you will need to call `shmget()` to create a new segment of the desired size. Alternatively, this call can be used to get the ID of a pre−existing shared memory segment. In either case, the return value is either the shared memory segment ID or −1 for error. For example, to create a shared memory segment of *b* bytes, the call might be `shmid = shmget(IPC_PRIVATE, b, (IPC_CREAT | 0666))`.
3. The next step is to attach this shared memory segment to this process, literally adding it to the virtual memory map of this process. Although the `shmat()` call allows the programmer to specify the virtual address at which the segment should appear, the address selected must be aligned on a page boundary (i.e., be a multiple of the page size returned by `getpagesize()`, which is usually 4096 bytes), and will override the mapping of any memory formerly at that address. Thus, we instead prefer to let the system pick the address. In either case, the return value is a pointer to the base virtual address of the segment just mapped. The code is `shmptr = shmat(shmid, 0, 0)`. Notice that you can allocate all your static shared variables into this shared memory segment by simply declaring all shared variables as members of a `struct` type, and declaring *shmptr* to be a pointer to that type. Using this technique, shared variable *x* would be accessed as *shmptr−>x*.
4. Since this shared memory segment should be destroyed when the last process with access to it terminates or detaches from it, we need to call `shmctl()` to set−up this default action. The code is something like `shmctl(shmid, IPC_RMID, 0)`.
5. Use the standard Linux `fork()` call to make the desired number of processes... each will inherit the shared memory segment.
6. When a process is done using a shared memory segment, it really should detach from that shared memory segment. This is done by `shmdt(shmptr)`.

Although the above set−up does require a few system calls, once the shared memory segment has been established, any change made by one processor to a value in that memory will automatically be visible to all processes. Most importantly, each communication operation will occur without the overhead of a system call.

An example C program using System V shared memory segments follows. It computes Pi, using the same algorithm given in section 1.3.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>

volatile struct shared { double pi; int lock; } *shared;

inline extern int xchg(register int reg,
volatile int * volatile obj)
{
  /* Atomic exchange instruction */
__asm__ __volatile__ ("xchgl %1,%0"
                      :"=r" (reg), "=m" (*obj)
                      :"r" (reg), "m" (*obj));
  return(reg);
}

main(int argc, char **argv)
{
  register double width, localsum;
  register int intervals, i;
  register int shmid;
  register int iproc = 0;;

  /* Allocate System V shared memory */
  shmid = shmget(IPC_PRIVATE,
                 sizeof(struct shared),
                 (IPC_CREAT | 0600));
  shared = ((volatile struct shared *) shmat(shmid, 0, 0));
  shmctl(shmid, IPC_RMID, 0);

  /* Initialize... */
  shared->pi = 0.0;
  shared->lock = 0;

  /* Fork a child */
  if (!fork()) ++iproc;

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  /* do the local computations */
  localsum = 0;
  for (i=iproc; i<intervals; i+=2) {
    register double x = (i + 0.5) * width;
    localsum += 4.0 / (1.0 + x * x);
  }
  localsum *= width;

  /* Atomic spin lock, add, unlock... */
  while (xchg((iproc + 1), &(shared->lock))) ;
  shared->pi += localsum;
```

```
  shared->lock = 0;

  /* Terminate child (barrier sync) */
  if (iproc == 0) {
    wait(NULL);
    printf("Estimation of pi is %f\n", shared->pi);
  }

  /* Check out */
  return(0);
}
```

In this example, I have used the IA32 atomic exchange instruction to implement locking. For better performance and portability, substitute a synchronization technique that avoids atomic bus−locking instructions (discussed in section 2.2).

When debugging your code, it is useful to remember that the `ipcs` command will report the status of the System V IPC facilities currently in use.

2.6 Memory Map Call

Using system calls for file I/O can be very expensive; in fact, that is why there is a user−buffered file I/O library (`getchar()`, `fwrite()`, etc.). But user buffers don't work if multiple processes are accessing the same writeable file, and the user buffer management overhead is significant. The BSD UNIX fix for this was the addition of a system call that allows a portion of a file to be mapped into user memory, essentially using virtual memory paging mechanisms to cause updates. This same mechanism also has been used in systems from Sequent for many years as the basis for their shared memory parallel processing support. Despite some very negative comments in the (quite old) man page, Linux seems to correctly perform at least some of the basic functions, and it supports the degenerate use of this system call to map an anonymous segment of memory that can be shared across multiple processes.

In essence, the Linux implementation of `mmap()` is a plug−in replacement for steps 2, 3, and 4 in the System V shared memory scheme outlined in section 2.5. To create an anonymous shared memory segment:

```
shmptr =
    mmap(0,                        /* system assigns address */
         b,                        /* size of shared memory segment */
         (PROT_READ | PROT_WRITE), /* access rights, can be rwx */
         (MAP_ANON | MAP_SHARED),  /* anonymous, shared */
         0,                        /* file descriptor (not used) */
         0);                       /* file offset (not used) */
```

The equivalent to the System V shared memory `shmdt()` call is `munmap()`:

```
munmap(shmptr, b);
```

In my opinion, there is no real benefit in using `mmap()` instead of the System V shared memory support.

3.Clusters Of Linux Systems

This section attempts to give an overview of cluster parallel processing using Linux. Clusters are currently both the most popular and the most varied approach, ranging from a conventional network of workstations (**NOW**) to essentially custom parallel machines that just happen to use Linux PCs as processor nodes. There is also quite a lot of software support for parallel processing using clusters of Linux machines.

3.1 Why A Cluster?

Cluster parallel processing offers several important advantages:

- Each of the machines in a cluster can be a complete system, usable for a wide range of other computing applications. This leads many people to suggest that cluster parallel computing can simply claim all the "wasted cycles" of workstations sitting idle on people's desks. It is not really so easy to salvage those cycles, and it will probably slow your co−worker's screen saver, but it can be done.
- The current explosion in networked systems means that most of the hardware for building a cluster is being sold in high volume, with correspondingly low "commodity" prices as the result. Further savings come from the fact that only one video card, monitor, and keyboard are needed for each cluster (although you may need to swap these into each machine to perform the initial installation of Linux, once running, a typical Linux PC does not need a "console"). In comparison, SMP and attached processors are much smaller markets, tending toward somewhat higher price per unit performance.
- Cluster computing can *scale to very large systems*. While it is currently hard to find a Linux−compatible SMP with many more than four processors, most commonly available network hardware easily builds a cluster with up to 16 machines. With a little work, hundreds or even thousands of machines can be networked. In fact, the entire Internet can be viewed as one truly huge cluster.
- The fact that replacing a "bad machine" within a cluster is trivial compared to fixing a partly faulty SMP yields much higher availability for carefully designed cluster configurations. This becomes important not only for particular applications that cannot tolerate significant service interruptions, but also for general use of systems containing enough processors so that single−machine failures are fairly common. (For example, even though the average time to failure of a PC might be two years, in a cluster with 32 machines, the probability that at least one will fail within 6 months is quite high.)

OK, so clusters are free or cheap and can be very large and highly available... why doesn't everyone use a cluster? Well, there are problems too:

- With a few exceptions, network hardware is not designed for parallel processing. Typically latency is very high and bandwidth relatively low compared to SMP and attached processors. For example, SMP latency is generally no more than a few microseconds, but is commonly hundreds or thousands of microseconds for a cluster. SMP communication bandwidth is often more than 100 MBytes/second; although the fastest network hardware (e.g., "Gigabit Ethernet") offers comparable speed, the most commonly used networks are between 10 and 1000 times slower. The performance of network hardware is poor enough as an *isolated cluster network*. If the network is not isolated from other traffic, as is often the case using "machines that happen to be networked" rather than a system designed as a cluster, performance can be substantially worse.

- There is very little software support for treating a cluster as a single system. For example, the `ps` command only reports the processes running on one Linux system, not all processes running across a cluster of Linux systems.

Thus, the basic story is that clusters offer great potential, but that potential may be very difficult to achieve for most applications. The good news is that there is quite a lot of software support that will help you achieve good performance for programs that are well suited to this environment, and there are also networks designed specifically to widen the range of programs that can achieve good performance.

3.2 Network Hardware

Computer networking is an exploding field... but you already knew that. An ever−increasing range of networking technologies and products are being developed, and most are available in forms that could be applied to make a parallel−processing cluster out of a group of machines (i.e., PCs each running Linux).

Unfortunately, no one network technology solves all problems best; in fact, the range of approach, cost, and performance is at first hard to believe. For example, using standard commercially−available hardware, the cost per machine networked ranges from less than $5 to over $4,000. The delivered bandwidth and latency each also vary over four orders of magnitude.

Before trying to learn about specific networks, it is important to recognize that these things change like the wind (see http://www.uk.linux.org/NetNews.html for Linux networking news), and it is very difficult to get accurate data about some networks.

Where I was particularly uncertain, I've placed a *?*. I have spent a lot of time researching this topic, but I'm sure my summary is full of errors and has omitted many important things. If you have any corrections or additions, please send email to pplinux@ecn.purdue.edu.

Summaries like the LAN Technology Scorecard at http://web.syr.edu/~jmwobus/comfaqs/lan−technology.html give some characteristics of many different types of networks and LAN standards. However, the summary in this HOWTO centers on the network properties that are most relevant to construction of Linux clusters. The section discussing each network begins with a short list of characteristics. The following defines what these entries mean.

Linux support:

If the answer is *no*, the meaning is pretty clear. Other answers try to describe the basic program interface that is used to access the network. Most network hardware is interfaced via a kernel driver, typically supporting TCP/UDP communication. Some other networks use more direct (e.g., library) interfaces to reduce latency by bypassing the kernel.

Years ago, it used to be considered perfectly acceptable to access a floating point unit via an OS call, but that is now clearly ludicrous; in my opinion, it is just as awkward for each communication between processors executing a parallel program to require an OS call. The problem is that computers haven't yet integrated these communication mechanisms, so non−kernel approaches tend to have portability problems. You are going to hear a lot more about this in the near future, mostly in the form of the new **Virtual Interface (VI) Architecture**, http://www.viarch.org/, which is a standardized method for most network interface operations to bypass the usual OS call layers. The VI standard is backed by Compaq, Intel, and Microsoft, and is sure to have a strong impact on SAN (System Area Network) designs over the next few years.

Maximum bandwidth:

This is the number everybody cares about. I have generally used the theoretical best case numbers; your mileage *will* vary.

Minimum latency:

In my opinion, this is the number everybody should care about even more than bandwidth. Again, I have used the unrealistic best−case numbers, but at least these numbers do include *all* sources of latency, both hardware and software. In most cases, the network latency is just a few microseconds; the much larger numbers reflect layers of inefficient hardware and software interfaces.

Available as:

Simply put, this describes how you get this type of network hardware. Commodity stuff is widely available from many vendors, with price as the primary distinguishing factor. Multiple−vendor things are available from more than one competing vendor, but there are significant differences and potential interoperability problems. Single−vendor networks leave you at the mercy of that supplier (however benevolent it may be). Public domain designs mean that even if you cannot find somebody to sell you one, you or anybody else can buy parts and make one. Research prototypes are just that; they are generally neither ready for external users nor available to them.

Interface port/bus used:

How does one hook−up this network? The highest performance and most common now is a PCI bus interface card. There are also EISA, VESA local bus (VL bus), and ISA bus cards. ISA was there first, and is still commonly used for low−performance cards. EISA is still around as the second bus in a lot of PCI machines, so there are a few cards. These days, you don't see much VL stuff (although http://www.vesa.org/ would beg to differ).
Of course, any interface that you can use without having to open your PC's case has more than a little appeal. IrDA and USB interfaces are appearing with increasing frequency. The Standard Parallel Port (SPP) used to be what your printer was plugged into, but it has seen a lot of use lately as an external extension of the ISA bus; this new functionality is enhanced by the IEEE 1284 standard, which specifies EPP and ECP improvements. There is also the old, reliable, slow RS232 serial port. I don't know of anybody connecting machines using VGA video connectors, keyboard, mouse, or game ports... so that's about it.

Network structure:

A bus is a wire, set of wires, or fiber. A hub is a little box that knows how to connect different wires/fibers plugged into it; switched hubs allow multiple connections to be actively transmitting data simultaneously.

Cost per machine connected:

Here's how to use these numbers. Suppose that, not counting the network connection, it costs \$2,000 to purchase a PC for use as a node in your cluster. Adding a Fast Ethernet brings the per node cost to about \$2,400; adding a Myrinet instead brings the cost to about \$3,800. If you have about \$20,000 to spend, that means you could have either 8 machines connected by Fast Ethernet or 5 machines connected by Myrinet. It also can be very reasonable to have multiple networks; e.g., \$20,000 could buy 8 machines connected by both Fast Ethernet and TTL_PAPERS. Pick the network, or set of networks, that is most likely to yield a cluster that will run your application fastest.
By the time you read this, these numbers will be wrong... heck, they're probably wrong already. There may also be quantity discounts, special deals, etc. Still, the prices quoted here aren't likely to be wrong enough to lead you to a totally inappropriate choice. It doesn't take a PhD (although I do have one ;−) to see that expensive networks only make sense if your application needs their special properties or if the PCs being clustered are relatively expensive.

Now that you have the disclaimers, on with the show....

ArcNet

- Linux support: *kernel drivers*
- Maximum bandwidth: *2.5 Mb/s*
- Minimum latency: *1,000 microseconds?*
- Available as: *multiple−vendor hardware*
- Interface port/bus used: *ISA*
- Network structure: *unswitched hub or bus (logical ring)*
- Cost per machine connected: *\$200*

ARCNET is a local area network that is primarily intended for use in embedded real−time control systems. Like Ethernet, the network is physically organized either as taps on a bus or one or more hubs, however, unlike Ethernet, it uses a token−based protocol logically structuring the network as a ring. Packet headers are small (3 or 4 bytes) and messages can carry as little as a single byte of data. Thus, ARCNET yields more consistent performance than Ethernet, with bounded delays, etc. Unfortunately, it is slower than Ethernet and less popular, making it more expensive. More information is available from the ARCNET Trade Association at http://www.arcnet.com/.

ATM

- Linux support: *kernel driver, AAL* library*
- Maximum bandwidth: *155 Mb/s* (soon, *1,200 Mb/s*)

* Minimum latency: *120 microseconds*
* Available as: *multiple−vendor hardware*
* Interface port/bus used: *PCI*
* Network structure: *switched hubs*
* Cost per machine connected: *$3,000*

Unless you've been in a coma for the past few years, you have probably heard a lot about how ATM (Asynchronous Transfer Mode) *is* the future... well, sort−of. ATM is cheaper than HiPPI and faster than Fast Ethernet, and it can be used over the very long distances that the phone companies care about. The ATM network protocol is also designed to provide a lower−overhead software interface and to more efficiently manage small messages and real−time communications (e.g., digital audio and video). It is also one of the highest−bandwidth networks that Linux currently supports. The bad news is that ATM isn't cheap, and there are still some compatibility problems across vendors. An overview of Linux ATM development is available at http://lrcwww.epfl.ch/linux−atm/.

CAPERS

* Linux support: *AFAPI library*
* Maximum bandwidth: *1.2 Mb/s*
* Minimum latency: *3 microseconds*
* Available as: *commodity hardware*
* Interface port/bus used: *SPP*
* Network structure: *cable between 2 machines*
* Cost per machine connected: *$2*

CAPERS (Cable Adapter for Parallel Execution and Rapid Synchronization) is a spin−off of the PAPERS project, http://garage.ecn.purdue.edu/~papers/, at the Purdue University School of Electrical and Computer Engineering. In essence, it defines a software protocol for using an ordinary "LapLink" SPP−to−SPP cable to implement the PAPERS library for two Linux PCs. The idea doesn't scale, but you can't beat the price. As with TTL_PAPERS, to improve system security, there is a minor kernel patch recommended, but not required: http://garage.ecn.purdue.edu/~papers/giveioperm.html.

Ethernet

* Linux support: *kernel drivers*
* Maximum bandwidth: *10 Mb/s*
* Minimum latency: *100 microseconds*
* Available as: *commodity hardware*
* Interface port/bus used: *PCI*
* Network structure: *switched or unswitched hubs, or hubless bus*
* Cost per machine connected: *$100* (hubless, *$50*)

For some years now, 10 Mbits/s Ethernet has been the standard network technology. Good Ethernet interface cards can be purchased for well under $50, and a fair number of PCs now have an Ethernet controller built−into the motherboard. For lightly−used networks, Ethernet connections can be organized as a multi−tap bus without a hub; such configurations can serve up to 200 machines with minimal cost, but are not appropriate for parallel processing. Adding an unswitched hub does not really help performance. However, switched hubs that can provide full bandwidth to simultaneous connections cost only about $100 per port. Linux supports an amazing range of Ethernet interfaces, but it is important to keep in mind that variations in the interface hardware can yield significant performance differences. See the Hardware Compatibility HOWTO for comments on which are supported and how well they work; also see http://cesdis1.gsfc.nasa.gov/linux/drivers/.

An interesting way to improve performance is offered by the 16−machine Linux cluster work done in the Beowulf project, http://cesdis.gsfc.nasa.gov/linux/beowulf/beowulf.html, at NASA CESDIS. There, Donald Becker, who is the author of many Ethernet card drivers, has developed support for load sharing across multiple Ethernet networks that shadow each other (i.e., share the same network addresses). This load sharing is built−into the standard Linux distribution, and is done invisibly below the socket operation level. Because hub cost is significant, having each machine connected to two or more hubless or unswitched hub Ethernet networks can be a very cost−effective way to improve performance. In fact, in situations where one machine is the network performance bottleneck, load sharing using shadow networks works much better than using a single switched hub network.

Ethernet (Fast Ethernet)

- Linux support: *kernel drivers*
- Maximum bandwidth: *100 Mb/s*
- Minimum latency: *80 microseconds*
- Available as: *commodity hardware*
- Interface port/bus used: *PCI*
- Network structure: *switched or unswitched hubs*
- Cost per machine connected: *$400?*

Although there are really quite a few different technologies calling themselves "Fast Ethernet," this term most often refers to a hub−based 100 Mbits/s Ethernet that is somewhat compatible with older "10 BaseT" 10 Mbits/s devices and cables. As might be expected, anything called Ethernet is generally priced for a volume market, and these interfaces are generally a small fraction of the price of 155 Mbits/s ATM cards. The catch is that having a bunch of machines dividing the bandwidth of a single 100 Mbits/s "bus" (using an unswitched hub) yields performance that might not even be as good on average as using 10 Mbits/s Ethernet with a switched hub that can give each machine's connection a full 10 Mbits/s.

Switched hubs that can provide 100 Mbits/s for each machine simultaneously are expensive, but prices are dropping every day, and these switches do yield much higher total network bandwidth than unswitched hubs. The thing that makes ATM switches so expensive is that they must switch for each (relatively short) ATM cell; some Fast Ethernet switches take advantage of the expected lower switching frequency by using techniques that may have low latency through the switch, but take multiple milliseconds to change the switch path... if your routing pattern changes frequently, avoid those switches. See http://cesdis1.gsfc.nasa.gov/linux/drivers/ for information about the various cards and drivers.

Also note that, as described for Ethernet, the Beowulf project, http://cesdis.gsfc.nasa.gov/linux/beowulf/beowulf.html, at NASA has been developing support that offers improved performance by load sharing across multiple Fast Ethernets.

Ethernet (Gigabit Ethernet)

- Linux support: *kernel drivers*
- Maximum bandwidth: *1,000 Mb/s*
- Minimum latency: *300 microseconds?*
- Available as: *multiple−vendor hardware*
- Interface port/bus used: *PCI*
- Network structure: *switched hubs or FDRs*
- Cost per machine connected: *$2,500?*

I'm not sure that Gigabit Ethernet, http://www.gigabit−ethernet.org/, has a good technological reason to be called Ethernet... but the name does accurately reflect the fact that this is intended to be a cheap, mass−market, computer network technology with native support for IP. However, current pricing reflects the fact that Gb/s hardware is still a tricky thing to build.

Unlike other Ethernet technologies, Gigabit Ethernet provides for a level of flow control that should make it a more reliable network. FDRs, or Full−Duplex Repeaters, simply multiplex lines, using buffering and localized flow control to improve performance. Most switched hubs are being built as new interface modules for existing gigabit−capable switch fabrics. Switch/FDR products have been shipped or announced by at least http://www.acacianet.com/, http://www.baynetworks.com/, http://www.cabletron.com/, http://www.networks.digital.com/, http://www.extremenetworks.com/, http://www.foundrynet.com/, http://www.gigalabs.com/, http://www.packetengines.com/. http://www.plaintree.com/, http://www.prominet.com/, http://www.sun.com/, and http://www.xlnt.com/.

There is a Linux driver, http://cesdis.gsfc.nasa.gov/linux/drivers/yellowfin.html, for the Packet Engines "Yellowfin" G−NIC, http://www.packetengines.com/. Early tests under Linux achieved about 2.5x higher bandwidth than could be achieved with the best 100 Mb/s Fast Ethernet; with gigabit networks, careful tuning of PCI bus use is a critical factor. There is little doubt that driver improvements, and Linux drivers for other NICs, will follow.

FC (Fibre Channel)

- Linux support: *no*
- Maximum bandwidth: *1,062 Mb/s*
- Minimum latency: *?*
- Available as: *multiple−vendor hardware*
- Interface port/bus used: *PCI?*
- Network structure: *?*
- Cost per machine connected: *?*

The goal of FC (Fibre Channel) is to provide high−performance block I/O (an FC frame carries a 2,048 byte data payload), particularly for sharing disks and other storage devices that can be directly connected to the FC rather than connected through a computer. Bandwidth−wise, FC is specified to be relatively fast, running anywhere between 133 and 1,062 Mbits/s. If FC becomes popular as a high−end SCSI replacement, it may quickly become a cheap technology; for now, it is not cheap and is not supported by Linux. A good collection of FC references is maintained by the Fibre Channel Association at http://www.amdahl.com/ext/CARP/FCA/FCA.html

FireWire (IEEE 1394)

- Linux support: *no*
- Maximum bandwidth: *196.608 Mb/s* (soon, *393.216 Mb/s*)
- Minimum latency: *?*
- Available as: *multiple−vendor hardware*
- Interface port/bus used: *PCI*
- Network structure: *random without cycles (self−configuring)*
- Cost per machine connected: *$600*

FireWire, http://www.firewire.org/, the IEEE 1394−1995 standard, is destined to be the low−cost high−speed digital network for consumer electronics. The showcase application is connecting DV digital video camcorders to computers, but FireWire is intended to be used for applications ranging from being a SCSI replacement to interconnecting the components of your home theater. It allows up to 64K devices to be connected in any topology using busses and bridges that does not create a cycle, and automatically detects the configuration when components are added or removed. Short (four−byte "quadlet") low−latency messages are supported as well as ATM−like isochronous transmission (used to keep multimedia messages synchronized). Adaptec has FireWire products that allow up to 63 devices to be connected to a single PCI interface card, and also has good general FireWire information at http://www.adaptec.com/serialio/.

Although FireWire will not be the highest bandwidth network available, the consumer−level market (which should drive prices very low) and low latency support might make this one of the best Linux PC cluster message−passing network technologies within the next year or so.

HiPPI And Serial HiPPI

- Linux support: *no*
- Maximum bandwidth: *1,600 Mb/s* (serial is *1,200 Mb/s*)
- Minimum latency: *?*
- Available as: *multiple−vendor hardware*
- Interface port/bus used: *EISA, PCI*
- Network structure: *switched hubs*
- Cost per machine connected: *$3,500* (serial is *$4,500*)

HiPPI (High Performance Parallel Interface) was originally intended to provide very high bandwidth for transfer of huge data sets between a supercomputer and another machine (a supercomputer, frame buffer, disk array, etc.), and has become the dominant standard for supercomputers. Although it is an oxymoron, **Serial**

HiPPI is also becoming popular, typically using a fiber optic cable instead of the 32−bit wide standard (parallel) HiPPI cables. Over the past few years, HiPPI crossbar switches have become common and prices have dropped sharply; unfortunately, serial HiPPI is still pricey, and that is what PCI bus interface cards generally support. Worse still, Linux doesn't yet support HiPPI. A good overview of HiPPI is maintained by CERN at http://www.cern.ch/HSI/hippi/; they also maintain a rather long list of HiPPI vendors at http://www.cern.ch/HSI/hippi/procintf/manufact.htm.

IrDA (Infrared Data Association)

- Linux support: *no?*
- Maximum bandwidth: *1.15 Mb/s* and *4 Mb/s*
- Minimum latency: *?*
- Available as: *multiple−vendor hardware*
- Interface port/bus used: *IrDA*
- Network structure: *thin air* ;−)
- Cost per machine connected: *$0*

IrDA (Infrared Data Association, http://www.irda.org/) is that little infrared device on the side of a lot of laptop PCs. It is inherently difficult to connect more than two machines using this interface, so it is unlikely to be used for clustering. Don Becker did some preliminary work with IrDA.

Myrinet

- Linux support: *library*
- Maximum bandwidth: *1,280 Mb/s*
- Minimum latency: *9 microseconds*
- Available as: *single−vendor hardware*
- Interface port/bus used: *PCI*
- Network structure: *switched hubs*
- Cost per machine connected: *$1,800*

Myrinet http://www.myri.com/ is a local area network (LAN) designed to also serve as a "system area network" (SAN), i.e., the network within a cabinet full of machines connected as a parallel system. The LAN and SAN versions use different physical media and have somewhat different characteristics; generally, the SAN version would be used within a cluster.

Myrinet is fairly conventional in structure, but has a reputation for being particularly well−implemented. The drivers for Linux are said to perform very well, although shockingly large performance variations have been reported with different PCI bus implementations for the host computers.

Currently, Myrinet is clearly the favorite network of cluster groups that are not too severely "budgetarily challenged." If your idea of a Linux PC is a high−end Pentium Pro or Pentium II with at least 256 MB RAM and a SCSI RAID, the cost of Myrinet is quite reasonable. However, using more ordinary PC configurations, you may find that your choice is between *N* machines linked by Myrinet or *2N* linked by multiple Fast

Ethernets and TTL_PAPERS. It really depends on what your budget is and what types of computations you care about most.

Parastation

- Linux support: *HAL or socket library*
- Maximum bandwidth: *125 Mb/s*
- Minimum latency: *2 microseconds*
- Available as: *single−vendor hardware*
- Interface port/bus used: *PCI*
- Network structure: *hubless mesh*
- Cost per machine connected: *> $1,000*

The ParaStation project http://wwwipd.ira.uka.de/parastation at University of Karlsruhe Department of Informatics is building a PVM−compatible custom low−latency network. They first constructed a two−processor ParaPC prototype using a custom EISA card interface and PCs running BSD UNIX, and then built larger clusters using DEC Alphas. Since January 1997, ParaStation has been available for Linux. The PCI cards are being made in cooperation with a company called Hitex (see http://www.hitex.com:80/parastation/). Parastation hardware implements both fast, reliable, message transmission and simple barrier synchronization.

PLIP

- Linux support: *kernel driver*
- Maximum bandwidth: *1.2 Mb/s*
- Minimum latency: *1,000 microseconds?*
- Available as: *commodity hardware*
- Interface port/bus used: *SPP*
- Network structure: *cable between 2 machines*
- Cost per machine connected: *$2*

For just the cost of a "LapLink" cable, PLIP (Parallel Line Interface Protocol) allows two Linux machines to communicate through standard parallel ports using standard socket−based software. In terms of bandwidth, latency, and scalability, this is not a very serious network technology; however, the near−zero cost and the software compatibility are useful. The driver is part of the standard Linux kernel distributions.

SCI

- Linux support: *no*
- Maximum bandwidth: *4,000 Mb/s*
- Minimum latency: *2.7 microseconds*

- Available as: *multiple−vendor hardware*
- Interface port/bus used: *PCI, proprietary*
- Network structure: *?*
- Cost per machine connected: *> $1,000*

The goal of SCI (Scalable Coherent Interconnect, ANSI/IEEE 1596−1992) is essentially to provide a high performance mechanism that can support coherent shared memory access across large numbers of machines, as well various types of block message transfers. It is fairly safe to say that the designed bandwidth and latency of SCI are both "awesome" in comparison to most other network technologies. The catch is that SCI is not widely available as cheap production units, and there isn't any Linux support.

SCI primarily is used in various proprietary designs for logically−shared physically−distributed memory machines, such as the HP/Convex Exemplar SPP and the Sequent NUMA−Q 2000 (see http://www.sequent.com/). However, SCI is available as a PCI interface card and 4−way switches (up to 16 machines can be connected by cascading four 4−way switches) from Dolphin, http://www.dolphinics.com/, as their CluStar product line. A good set of links overviewing SCI is maintained by CERN at http://www.cern.ch/HSI/sci/sci.html.

SCSI

- Linux support: *kernel drivers*
- Maximum bandwidth: *5 Mb/s* to over *20 Mb/s*
- Minimum latency: *?*
- Available as: *multiple−vendor hardware*
- Interface port/bus used: *PCI, EISA, ISA card*
- Network structure: *inter−machine bus sharing SCSI devices*
- Cost per machine connected: *?*

SCSI (Small Computer Systems Interconnect) is essentially an I/O bus that is used for disk drives, CD ROMS, image scanners, etc. There are three separate standards SCSI−1, SCSI−2, and SCSI−3; Fast and Ultra speeds; and data path widths of 8, 16, or 32 bits (with FireWire compatibility also mentioned in SCSI−3). It is all pretty confusing, but we all know a good SCSI is somewhat faster than EIDE and can handle more devices more efficiently.

What many people do not realize is that it is fairly simple for two computers to share a single SCSI bus. This type of configuration is very useful for sharing disk drives between machines and implementing **fail−over** − having one machine take over database requests when the other machine fails. Currently, this is the only mechanism supported by Microsoft's PC cluster product, WolfPack. However, the inability to scale to larger systems renders shared SCSI uninteresting for parallel processing in general.

ServerNet

- Linux support: *no*
- Maximum bandwidth: *400 Mb/s*

- Minimum latency: *3 microseconds*
- Available as: *single−vendor hardware*
- Interface port/bus used: *PCI*
- Network structure: *hexagonal tree/tetrahedral lattice of hubs*
- Cost per machine connected: *?*

ServerNet is the high−performance network hardware from Tandem, http://www.tandem.com. Especially in the online transation processing (OLTP) world, Tandem is well known as a leading producer of high−reliability systems, so it is not surprising that their network claims not just high performance, but also "high data integrity and reliability." Another interesting aspect of ServerNet is that it claims to be able to transfer data from any device directly to any device; not just between processors, but also disk drives, etc., in a one−sided style similar to that suggested by the MPI remote memory access mechanisms described in section 3.5. One last comment about ServerNet: although there is just a single vendor, that vendor is powerful enough to potentially establish ServerNet as a major standard... Tandem is owned by Compaq.

SHRIMP

- Linux support: *user−level memory mapped interface*
- Maximum bandwidth: *180 Mb/s*
- Minimum latency: *5 microseconds*
- Available as: *research prototype*
- Interface port/bus used: *EISA*
- Network structure: *mesh backplane (as in Intel Paragon)*
- Cost per machine connected: *?*

The SHRIMP project, http://www.CS.Princeton.EDU/shrimp/, at the Princeton University Computer Science Department is building a parallel computer using PCs running Linux as the processing elements. The first SHRIMP (Scalable, High−Performance, Really Inexpensive Multi−Processor) was a simple two−processor prototype using a dual−ported RAM on a custom EISA card interface. There is now a prototype that will scale to larger configurations using a custom interface card to connect to a "hub" that is essentially the same mesh routing network used in the Intel Paragon (see http://www.ssd.intel.com/paragon.html). Considerable effort has gone into developing low−overhead "virtual memory mapped communication" hardware and support software.

SLIP

- Linux support: *kernel drivers*
- Maximum bandwidth: *0.1 Mb/s*
- Minimum latency: *1,000 microseconds?*
- Available as: *commodity hardware*
- Interface port/bus used: *RS232C*
- Network structure: *cable between 2 machines*
- Cost per machine connected: *$2*

Although SLIP (Serial Line Interface Protocol) is firmly planted at the low end of the performance spectrum, SLIP (or CSLIP or PPP) allows two machines to perform socket communication via ordinary RS232 serial ports. The RS232 ports can be connected using a null−modem RS232 serial cable, or they can even be connected via dial−up through a modem. In any case, latency is high and bandwidth is low, so SLIP should be used only when no other alternatives are available. It is worth noting, however, that most PCs have two RS232 ports, so it would be possible to network a group of machines simply by connecting the machines as a linear array or as a ring. There is even load sharing software called EQL.

TTL_PAPERS

- Linux support: *AFAPI library*
- Maximum bandwidth: *1.6 Mb/s*
- Minimum latency: *3 microseconds*
- Available as: *public−domain design, single−vendor hardware*
- Interface port/bus used: *SPP*
- Network structure: *tree of hubs*
- Cost per machine connected: *$100*

The PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization) project, http://garage.ecn.purdue.edu/~papers/, at the Purdue University School of Electrical and Computer Engineering is building scalable, low−latency, aggregate function communication hardware and software that allows a parallel supercomputer to be built using unmodified PCs/workstations as nodes.

There have been over a dozen different types of PAPERS hardware built that connect to PCs/workstations via the SPP (Standard Parallel Port), roughly following two development lines. The versions called "PAPERS" target higher performance, using whatever technologies are appropriate; current work uses FPGAs, and high bandwidth PCI bus interface designs are also under development. In contrast, the versions called "TTL_PAPERS" are designed to be easily reproduced outside Purdue, and are remarkably simple public domain designs that can be built using ordinary TTL logic. One such design is produced commercially, http://chelsea.ios.com:80/~hgdietz/sbm4.html.

Unlike the custom hardware designs from other universities, TTL_PAPERS clusters have been assembled at many universities from the USA to South Korea. Bandwidth is severely limited by the SPP connections, but PAPERS implements very low latency aggregate function communications; even the fastest message−oriented systems cannot provide comparable performance on those aggregate functions. Thus, PAPERS is particularly good for synchronizing the displays of a video wall (to be discussed further in the upcoming Video Wall HOWTO), scheduling accesses to a high−bandwidth network, evaluating global fitness in genetic searches, etc. Although PAPERS clusters have been built using IBM PowerPC AIX, DEC Alpha OSF/1, and HP PA−RISC HP−UX machines, Linux−based PCs are the platforms best supported.

User programs using TTL_PAPERS AFAPI directly access the SPP hardware port registers under Linux, without an OS call for each access. To do this, AFAPI first gets port permission using either `iopl()` or `ioperm()`. The problem with these calls is that both require the user program to be privileged, yielding a potential security hole. The solution is an optional kernel patch, http://garage.ecn.purdue.edu/~papers/giveioperm.html, that allows a privileged process to control port permission for any process.

USB (Universal Serial Bus)

- Linux support: *kernel driver*
- Maximum bandwidth: *12 Mb/s*
- Minimum latency: *?*
- Available as: *commodity hardware*
- Interface port/bus used: *USB*
- Network structure: *bus*
- Cost per machine connected: *$5?*

USB (Universal Serial Bus, http://www.usb.org/) is a hot−pluggable conventional−Ethernet−speed, bus for up to 127 peripherals ranging from keyboards to video conferencing cameras. It isn't really clear how multiple computers get connected to each other using USB. In any case, USB ports are quickly becoming as standard on PC motherboards as RS232 and SPP, so don't be surprised if one or two USB ports are lurking on the back of the next PC you buy. Development of a Linux driver is discussed at http://peloncho.fis.ucm.es/~inaky/USB.html.

In some ways, USB is almost the low−performance, zero−cost, version of FireWire that you can purchase today.

WAPERS

- Linux support: *AFAPI library*
- Maximum bandwidth: *0.4 Mb/s*
- Minimum latency: *3 microseconds*
- Available as: *public−domain design*
- Interface port/bus used: *SPP*
- Network structure: *wiring pattern between 2−64 machines*
- Cost per machine connected: *$5*

WAPERS (Wired−AND Adapter for Parallel Execution and Rapid Synchronization) is a spin−off of the PAPERS project, http://garage.ecn.purdue.edu/~papers/, at the Purdue University School of Electrical and Computer Engineering. If implemented properly, the SPP has four bits of open−collector output that can be wired together across machines to implement a 4−bit wide wired AND. This wired−AND is electrically touchy, and the maximum number of machines that can be connected in this way critically depends on the analog properties of the ports (maximum sink current and pull−up resistor value); typically, up to 7 or 8 machines can be networked by WAPERS. Although cost and latency are very low, so is bandwidth; WAPERS is much better as a second network for aggregate operations than as the only network in a cluster. As with TTL_PAPERS, to improve system security, there is a minor kernel patch recommended, but not required: http://garage.ecn.purdue.edu/~papers/giveioperm.html.

3.3 Network Software Interface

Before moving on to discuss the software support for parallel applications, it is useful to first briefly cover the basics of low−level software interface to the network hardware. There are really only three basic choices: sockets, device drivers, and user−level libraries.

Sockets

By far the most common low−level network interface is a socket interface. Sockets have been a part of unix for over a decade, and most standard network hardware is designed to support at least two types of socket protocols: UDP and TCP. Both types of socket allow you to send arbitrary size blocks of data from one machine to another, but there are several important differences. Typically, both yield a minimum latency of around 1,000 microseconds, although performance can be far worse depending on network traffic.

These socket types are the basic network software interface for most of the portable, higher−level, parallel processing software; for example, PVM uses a combination of UDP and TCP, so knowing the difference will help you tune performance. For even better performance, you can also use these mechanisms directly in your program. The following is just a simple overview of UDP and TCP; see the manual pages and a good network programming book for details.

UDP Protocol (SOCK_DGRAM)

UDP is the User Datagram Protocol, but you more easily can remember the properties of UDP as Unreliable Datagram Processing. In other words, UDP allows each block to be sent as an individual message, but a message might be lost in transmission. In fact, depending on network traffic, UDP messages can be lost, can arrive multiple times, or can arrive in an order different from that in which they were sent. The sender of a UDP message does not automatically get an acknowledgment, so it is up to user−written code to detect and compensate for these problems. Fortunately, UDP does ensure that if a message arrives, the message contents are intact (i.e., you never get just part of a UDP message).

The nice thing about UDP is that it tends to be the fastest socket protocol. Further, UDP is "connectionless," which means that each message is essentially independent of all others. A good analogy is that each message is like a letter to be mailed; you might send multiple letters to the same address, but each one is independent of the others and there is no limit on how many people you can send letters to.

TCP Protocol (SOCK_STREAM)

Unlike UDP, **TCP** is a reliable, connection−based, protocol. Each block sent is not seen as a message, but as a block of data within an apparently continuous stream of bytes being transmitted through a connection between sender and receiver. This is very different from UDP messaging because each block is simply part of the byte stream and it is up to the user code to figure−out how to extract each block from the byte stream; there are no markings separating messages. Further, the connections are more fragile with respect to network problems, and only a limited number of connections can exist simultaneously for each process. Because it is reliable, TCP generally implies significantly more overhead than UDP.

There are, however, a few pleasant surprises about TCP. One is that, if multiple messages are sent through a connection, TCP is able to pack them together in a buffer to better match network hardware packet sizes, potentially yielding better−than−UDP performance for groups of short or oddly−sized messages. The other bonus is that networks constructed using reliable direct physical links between machines can easily and efficiently simulate TCP connections. For example, this was done for the ParaStation's "Socket Library" interface software, which provides TCP semantics using user−level calls that differ from the standard TCP OS calls only by the addition of the prefix `PSS` to each function name.

Device Drivers

When it comes to actually pushing data onto the network or pulling data off the network, the standard unix software interface is a part of the unix kernel called a device driver. UDP and TCP don't just transport data, they also imply a fair amount of overhead for socket management. For example, something has to manage the fact that multiple TCP connections can share a single physical network interface. In contrast, a device driver for a dedicated network interface only needs to implement a few simple data transport functions. These device driver functions can then be invoked by user programs by using `open()` to identify the proper device and then using system calls like `read()` and `write()` on the open "file." Thus, each such operation could transport a block of data with little more than the overhead of a system call, which might be as fast as tens of microseconds.

Writing a device driver to be used with Linux is not hard... if you know *precisely* how the device hardware works. If you are not sure how it works, don't guess. Debugging device drivers isn't fun and mistakes can fry hardware. However, if that hasn't scared you off, it may be possible to write a device driver to, for example, use dedicated Ethernet cards as dumb but fast direct machine−to−machine connections without the usual Ethernet protocol overhead. In fact, that's pretty much what some early Intel supercomputers did.... Look at the Device Driver HOWTO for more information.

User−Level Libraries

If you've taken an OS course, user−level access to hardware device registers is exactly what you have been taught never to do, because one of the primary purposes of an OS is to control device access. However, an OS call is at least tens of microseconds of overhead. For custom network hardware like TTL_PAPERS, which can perform a basic network operation in just 3 microseconds, such OS call overhead is intolerable. The only way to avoid that overhead is to have user−level code − a user−level library − directly access hardware device registers. Thus, the question becomes one of how a user−level library can access hardware directly, yet not compromise the OS control of device access rights.

On a typical system, the only way for a user−level library to directly access hardware device registers is to:

1. At user program start−up, use an OS call to map the page of memory address space containing the device registers into the user process virtual memory map. For some systems, the `mmap()` call (first mentioned in section 2.6) can be used to map a special file which represents the physical memory

page addresses of the I/O devices. Alternatively, it is relatively simple to write a device driver to perform this function. Further, this device driver can control access by only mapping the page(s) containing the specific device registers needed, thereby maintaining OS access control.
2. Access device registers without an OS call by simply loading or storing to the mapped addresses. For example, `*((char *) 0x1234) = 5;` would store the byte value 5 into memory location 1234 (hexadecimal).

Fortunately, it happens that Linux for the Intel 386 (and compatible processors) offers an even better solution:

1. Using the `ioperm()` OS call from a privileged process, get permission to access the precise I/O port addresses that correspond to the device registers. Alternatively, permission can be managed by an independent privileged user process (i.e., a "meta OS") using the giveioperm() OS call patch for Linux.
2. Access device registers without an OS call by using 386 port I/O instructions.

This second solution is preferable because it is common that multiple I/O devices have their registers within a single page, in which case the first technique would not provide protection against accessing other device registers that happened to reside in the same page as the ones intended. Of course, the down side is that 386 port I/O instructions cannot be coded in C − instead, you will need to use a bit of assembly code. The GCC−wrapped (usable in C programs) inline assembly code function for a port input of a byte value is:

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

Similarly, the GCC−wrapped code for a byte port output is:

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

3.4 PVM (Parallel Virtual Machine)

PVM (Parallel Virtual Machine) is a freely−available, portable, message−passing library generally implemented on top of sockets. It is clearly established as the de−facto standard for message−passing cluster parallel computing.

PVM supports single−processor and SMP Linux machines, as well as clusters of Linux machines linked by socket−capable networks (e.g., SLIP, PLIP, Ethernet, ATM). In fact, PVM will even work across groups of machines in which a variety of different types of processors, configurations, and physical networks are used − **Heterogeneous Clusters** − even to the scale of treating machines linked by the Internet as a parallel cluster. PVM also provides facilities for parallel job control across a cluster. Best of all, PVM has long been freely available (currently from http://www.epm.ornl.gov/pvm/pvm_home.html), which has led to many programming language compilers, application libraries, programming and debugging tools, etc., using it as their "portable message−passing target library." There is also a network newsgroup, comp.parallel.pvm.

It is important to note, however, that PVM message−passing calls generally add significant overhead to standard socket operations, which already had high latency. Further, the message handling calls themselves do not constitute a particularly "friendly" programming model.

Using the same Pi computation example first described in section 1.3, the version using C with PVM library calls is:

```
#include <stdlib.h>
#include <stdio.h>
#include <pvm3.h>

#define NPROC   4

main(int argc, char **argv)
{
  register double lsum, width;
  double sum;
  register int intervals, i;
  int mytid, iproc, msgtag = 4;
  int tids[NPROC];  /* array of task ids */

  /* enroll in pvm */
  mytid = pvm_mytid();

  /* Join a group and, if I am the first instance,
     iproc=0, spawn more copies of myself
  */
  iproc = pvm_joingroup("pi");

  if (iproc == 0) {
    tids[0] = pvm_mytid();
    pvm_spawn("pvm_pi", &argv[1], 0, NULL, NPROC-1, &tids[1]);
  }
  /* make sure all processes are here */
  pvm_barrier("pi", NPROC);

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;
```

```
  lsum = 0.0;
  for (i = iproc; i<intervals; i+=NPROC) {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }

  /* sum across the local results & scale by width */
  sum = lsum * width;
  pvm_reduce(PvmSum, &sum, 1, PVM_DOUBLE, msgtag, "pi", 0);

  /* have only the console PE print the result */
  if (iproc == 0) {
    printf("Estimation of pi is %f\n", sum);
  }

  /* Check program finished, leave group, exit pvm */
  pvm_barrier("pi", NPROC);
  pvm_lvgroup("pi");
  pvm_exit();
  return(0);
}
```

3.5 MPI (Message Passing Interface)

Although PVM is the de−facto standard message−passing library, MPI (Message Passing Interface) is the relatively new official standard. The home page for the MPI standard is http://www.mcs.anl.gov:80/mpi/ and the newsgroup is comp.parallel.mpi.

However, before discussing MPI, I feel compelled to say a little bit about the PVM vs. MPI religious war that has been going on for the past few years. I'm not really on either side. Here's my attempt at a relatively unbiased summary of the differences:

Execution control environment.

Put simply, PVM has one and MPI doesn't specify how/if one is implemented. Thus, things like starting a PVM program executing are done identically everywhere, while for MPI it depends on which implementation is being used.

Support for heterogeneous clusters.

PVM grew−up in the workstation cycle−scavenging world, and thus directly manages heterogeneous mixes of machines and operating systems. In contrast, MPI largely assumes that the target is an MPP (Massively Parallel Processor) or a dedicated cluster of nearly identical workstations.

Kitchen sink syndrome.

PVM evidences a unity of purpose that MPI 2.0 doesn't. The new MPI 2.0 standard includes a lot of features that go way beyond the basic message passing model − things like RMA

(Remote Memory Access) and parallel file I/O. Are these things useful? Of course they are... but learning MPI 2.0 is a lot like learning a complete new programming language.

User interface design.

MPI was designed after PVM, and clearly learned from it. MPI offers simpler, more efficient, buffer handling and higher−level abstractions allowing user−defined data structures to be transmitted in messages.

The force of law.

By my count, there are still significantly more things designed to use PVM than there are to use MPI; however, porting them to MPI is easy, and the fact that MPI is backed by a widely−supported formal standard means that using MPI is, for many institutions, a matter of policy.

Conclusion? Well, there are at least three independently developed, freely available, versions of MPI that can run on clusters of Linux systems (and I wrote one of them):

- LAM (Local Area Multicomputer) is a full implementation of the MPI 1.1 standard. It allows MPI programs to be executed within an individual Linux system or across a cluster of Linux systems using UDP/TCP socket communication. The system includes simple execution control facilities, as well as a variety of program development and debugging aids. It is freely available from http://www.osc.edu/lam.html.
- MPICH (MPI CHameleon) is designed as a highly portable full implementation of the MPI 1.1 standard. Like LAM, it allows MPI programs to be executed within an individual Linux system or across a cluster of Linux systems using UDP/TCP socket communication. However, the emphasis is definitely on promoting MPI by providing an efficient, easily retargetable, implementation. To port this MPI implementation, one implements either the five functions of the "channel interface" or, for better performance, the full MPICH ADI (Abstract Device Interface). MPICH, and lots of information about it and porting, are available from http://www.mcs.anl.gov/mpi/mpich/.
- AFMPI (Aggregate Function MPI) is a subset implementation of the MPI 2.0 standard. This is the one that I wrote. Built on top of the AFAPI, it is designed to showcase low−latency collective communication functions and RMAs, and thus provides only minimal support for MPI data types, communicators, etc. It allows C programs using MPI to run on an individual Linux system or across a cluster connected by AFAPI−capable network hardware. It is freely available from http://garage.ecn.purdue.edu/~papers/.

No matter which of these (or other) MPI implementations one uses, it is fairly simple to perform the most common types of communications.

However, MPI 2.0 incorporates several communication paradigms that are fundamentally different enough so that a programmer using one of them might not even recognize the other coding styles as MPI. Thus, rather than giving a single example program, it is useful to have an example of each of the fundamentally different communication paradigms that MPI supports. All three programs implement the same basic algorithm (from section 1.3) that is used throughout this HOWTO to compute the value of Pi.

The first MPI program uses basic MPI message−passing calls for each processor to send its partial sum to processor 0, which sums and prints the result:

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  register double width;
  double sum, lsum;
  register int intervals, i;
  int nproc, iproc;
  MPI_Status status;

  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;
  lsum = 0;
  for (i=iproc; i<intervals; i+=nproc) {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
  lsum *= width;
  if (iproc != 0) {
    MPI_Send(&lbuf, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  } else {
    sum = lsum;
    for (i=1; i<nproc; ++i) {
      MPI_Recv(&lbuf, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
               MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      sum += lsum;
    }
    printf("Estimation of pi is %f\n", sum);
  }
  MPI_Finalize();
  return(0);
}
```

The second MPI version uses collective communication (which, for this particular application, is clearly the most appropriate):

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  register double width;
  double sum, lsum;
  register int intervals, i;
  int nproc, iproc;

  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;
  lsum = 0;
```

```
  for (i=iproc; i<intervals; i+=nproc) {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
  lsum *= width;
  MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
             MPI_SUM, 0, MPI_COMM_WORLD);
  if (iproc == 0) {
    printf("Estimation of pi is %f\n", sum);
  }
  MPI_Finalize();
  return(0);
}
```

The third MPI version uses the MPI 2.0 RMA mechanism for each processor to add its local `lsum` into `sum` on processor 0:

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  register double width;
  double sum = 0, lsum;
  register int intervals, i;
  int nproc, iproc;
  MPI_Win sum_win;

  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  MPI_Win_create(&sum, sizeof(sum), sizeof(sum),
                 0, MPI_COMM_WORLD, &sum_win);
  MPI_Win_fence(0, sum_win);
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;
  lsum = 0;
  for (i=iproc; i<intervals; i+=nproc) {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
  lsum *= width;
  MPI_Accumulate(&lsum, 1, MPI_DOUBLE, 0, 0,
                 1, MPI_DOUBLE, MPI_SUM, sum_win);
  MPI_Win_fence(0, sum_win);
  if (iproc == 0) {
    printf("Estimation of pi is %f\n", sum);
  }
  MPI_Finalize();
  return(0);
}
```

It is useful to note that the MPI 2.0 RMA mechanism very neatly overcomes any potential problems with the corresponding data structure on various processors residing at different memory locations. This is done by referencing a "window" that implies the base address, protection against out−of−bound accesses, and even address scaling. Efficient implementation is aided by the fact that RMA processing may be delayed until the next `MPI_Win_fence`. In summary, the RMA mechanism may be a strange cross between distributed shared memory and message passing, but it is a very clean interface that potentially generates very efficient communication.

3.6 AFAPI (Aggregate Function API)

Unlike PVM, MPI, etc., the AFAPI (Aggregate Function Application Program Interface) did not start life as an attempt to build a portable abstract interface layered on top of existing network hardware and software. Rather, AFAPI began as the very hardware−specific low−level support library for PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization; see http://garage.ecn.purdue.edu/~papers/).

PAPERS was discussed briefly in section 3.2; it is a public domain design custom aggregate function network that delivers latencies as low as a few microseconds. However, the important thing about PAPERS is that it was developed as an attempt to build a supercomputer that would be a better target for compiler technology than existing supercomputers. This is qualitatively different from most Linux cluster efforts and PVM/MPI, which generally focus on trying to use standard networks for the relatively few sufficiently coarse−grain parallel applications. The fact that Linux PCs are used as components of PAPERS systems is simply an artifact of implementing prototypes in the most cost−effective way possible.

The need for a common low−level software interface across more than a dozen different prototype implementations was what made the PAPERS library become standardized as AFAPI. However, the model used by AFAPI is inherently simpler and better suited for the finer−grain interactions typical of code compiled by parallelizing compilers or written for SIMD architectures. The simplicity of the model not only makes PAPERS hardware easy to build, but also yields surprisingly efficient AFAPI ports for a variety of other hardware systems, such as SMPs.

AFAPI currently runs on Linux clusters connected using TTL_PAPERS, CAPERS, or WAPERS. It also runs (without OS calls or even bus−lock instructions, see section 2.2) on SMP systems using a System V Shared Memory library called SHMAPERS. A version that runs across Linux clusters using UDP broadcasts on conventional networks (e.g., Ethernet) is under development. All released versions are available from http://garage.ecn.purdue.edu/~papers/. All versions of the AFAPI are designed to be called from C or C++.

The following example program is the AFAPI version of the Pi computation described in section 1.3.

```
#include <stdlib.h>
#include <stdio.h>
#include "afapi.h"

main(int argc, char **argv)
{
  register double width, sum;
  register int intervals, i;

  if (p_init()) exit(1);

  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  sum = 0;
  for (i=IPROC; i<intervals; i+=NPROC) {
    register double x = (i + 0.5) * width;
```

```
    sum += 4.0 / (1.0 + x * x);
  }

  sum = p_reduceAdd64f(sum) * width;

  if (IPROC == CPROC) {
    printf("Estimation of pi is %f\n", sum);
  }

  p_exit();
  return(0);
}
```

3.7 Other Cluster Support Libraries

In addition to PVM, MPI, and AFAPI, the following libraries offer features that may be useful in parallel computing using a cluster of Linux systems. These systems are given a lighter treatment in this document simply because, unlike PVM, MPI, and AFAPI, I have little or no direct experience with the use of these systems on Linux clusters. If you find any of these or other libraries to be especially useful, please send email to me at pplinux@ecn.purdue.edu describing what you've found, and I will consider adding an expanded section on that library.

Condor (process migration support)

Condor is a distributed resource management system that can manage large heterogeneous clusters of workstations. Its design has been motivated by the needs of users who would like to use the unutilized capacity of such clusters for their long−running, computation−intensive jobs. Condor preserves a large measure of the originating machine's environment on the execution machine, even if the originating and execution machines do not share a common file system and/or password mechanisms. Condor jobs that consist of a single process are automatically checkpointed and migrated between workstations as needed to ensure eventual completion.

Condor is available at http://www.cs.wisc.edu/condor/. A Linux port exists; more information is available at http://www.cs.wisc.edu/condor/linux/linux.html. Contact condor−admin@cs.wisc.edu for details.

DFN−RPC (German Research Network − Remote Procedure Call)

The DFN−RPC, a (German Research Network Remote Procedure Call) tool, was developed to distribute and parallelize scientific−technical application programs between a workstation and a compute server or a cluster. The interface is optimized for applications written in fortran, but the DFN−RPC can also be used in a C environment. It has been ported to Linux. More information is at ftp://ftp.uni−stuttgart.de/pub/rus/dfn_rpc/README_dfnrpc.html.

DQS (Distributed Queueing System)

Not exactly a library, DQS 3.0 (Distributed Queueing System) is a job queueing system that has been developed and tested under Linux. It is designed to allow both use and administration of a heterogeneous cluster as a single entity. It is available from http://www.scri.fsu.edu/~pasko/dqs.html.

There is also a commercial version called CODINE 4.1.1 (COmputing in DIstributed Network Environments). Information on it is available from http://www.genias.de/genias_welcome.html.

3.8 General Cluster References

Because clusters can be constructed and used in so many different ways, there are quite a few groups that have made interesting contributions. The following are references to various cluster−related projects that may be of general interest. This includes a mix of Linux−specific and generic cluster references. The list is given in alphabetical order.

Beowulf

The Beowulf project, http://cesdis1.gsfc.nasa.gov/beowulf/, centers on production of software for using off−the−shelf clustered workstations based on commodity PC−class hardware, a high−bandwidth cluster−internal network, and the Linux operating system.

Thomas Sterling has been the driving force behind Beowulf, and continues to be an eloquent and outspoken proponent of Linux clustering for scientific computing in general. In fact, many groups now refer to their clusters as "Beowulf class" systems − even if the cluster isn't really all that similar to the official Beowulf design.

Don Becker, working in support of the Beowulf project, has produced many of the network drivers used by Linux in general. Many of these drivers have even been adapted for use in BSD. Don also is responsible for many of these Linux network drivers allowing load−sharing across multiple parallel connections to achieve higher bandwidth without expensive switched hubs. This type of load sharing was the original distinguishing feature of the Beowulf cluster.

Linux/AP+

The Linux/AP+ project, http://cap.anu.edu.au/cap/projects/linux/, is not exactly about Linux clustering, but centers on porting Linux to the Fujitsu AP1000+ and adding appropriate parallel processing enhancements. The AP1000+ is a commercially available SPARC−based parallel machine that uses a custom network with a torus topology, 25 MB/s bandwidth, and 10 microsecond latency... in short, it looks a lot like a SPARC Linux cluster.

Locust

The Locust project, http://www.ecsl.cs.sunysb.edu/~manish/locust/, is building a distributed virtual shared memory system that uses compile−time information to hide message−latency and to reduce network traffic at run time. Pupa is the underlying communication subsystem of Locust, and is implemented using Ethernet to connect 486 PCs under FreeBSD. Linux?

Midway DSM (Distributed Shared Memory)

Midway, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/midway/WWW/HomePage.html, is a software−based DSM (Distributed Shared Memory) system, not unlike TreadMarks. The good news is that it uses compile−time aids rather than relatively slow page−fault mechanisms, and it is free. The bad news is that it doesn't run on Linux clusters.

Mosix

MOSIX modifies the BSDI BSD/OS to provide dynamic load balancing and preemptive process migration across a networked group of PCs. This is nice stuff not just for parallel processing, but for generally using a cluster much like a scalable SMP. Will there be a Linux version? Look at http://www.cs.huji.ac.il/mosix/ for more information.

NOW (Network Of Workstations)

The Berkeley NOW (Network Of Workstations) project, http://now.cs.berkeley.edu/, has led much of the push toward parallel computing using networks of workstations. There is a lot work going on here, all aimed toward "demonstrating a practical 100 processor system in the next few years." Alas, they don't use Linux.

Parallel Processing Using Linux

The parallel processing using Linux WWW site, http://yara.ecn.purdue.edu/~pplinux/, is the home of this HOWTO and many related documents including online slides for a full−day tutorial. Aside from the work on the PAPERS project, the Purdue University School of Electrical and Computer Engineering generally has been a leader in parallel processing; this site was established to help others apply Linux PCs for parallel processing.

Since Purdue's first cluster of Linux PCs was assembled in February 1994, there have been many Linux PC clusters assembled at Purdue, including several with video walls. Although these clusters used 386, 486, and Pentium systems (no Pentium Pro systems), Intel recently awarded Purdue a donation which will allow it to construct multiple large clusters of Pentium II systems (with as many as 165 machines planned for a single cluster). Although these clusters all have/will have PAPERS networks, most also have conventional networks.

Pentium Pro Cluster Workshop

In Des Moines, Iowa, April 10−11, 1997, AMES Laboratory held the Pentium Pro Cluster Workshop. The WWW site from this workshop, http://www.scl.ameslab.gov/workshops/PPCworkshop.html, contains a wealth of PC cluster information gathered from all the attendees.

TreadMarks DSM (Distributed Shared Memory)

DSM (Distributed Shared Memory) is a technique whereby a message−passing system can appear to behave as an SMP. There are quite a few such systems, most of which use the OS page−fault mechanism to trigger message transmissions. TreadMarks, http://www.cs.rice.edu/~willy/TreadMarks/overview.html, is one of the more efficient of such systems and does run on Linux clusters. The bad news is "TreadMarks is being distributed at a small cost to universities and nonprofit institutions." For more information about the software, contact treadmarks@ece.rice.edu.

U−Net (User−level NETwork interface architecture)

The U−Net (User−level NETwork interface architecture) project at Cornell, http://www2.cs.cornell.edu/U−Net/Default.html, attempts to provide low−latency and high−bandwidth using commodity network hardware by by virtualizing the network interface so that applications can send and receive messages without operating system intervention. U−Net runs on Linux PCs using a DECchip DC21140 based Fast Ethernet card or a Fore Systems PCA−200 (not PCA−200E) ATM card.

WWT (Wisconsin Wind Tunnel)

There is really quite a lot of cluster−related work at Wisconsin. The WWT (Wisconsin Wind Tunnel) project, http://www.cs.wisc.edu/~wwt/, is doing all sorts of work toward developing a "standard" interface between compilers and the underlying parallel hardware. There is the Wisconsin COW (Cluster Of Workstations), Cooperative Shared Memory and Tempest, the Paradyn Parallel Performance Tools, etc. Unfortunately, there is not much about Linux.

4.SIMD Within A Register (e.g., using MMX)

SIMD (Single Instruction stream, Multiple Data stream) Within A Register (SWAR) isn't a new idea. Given a machine with *k*−bit registers, data paths, and function units, it has long been known that ordinary register operations can function as SIMD parallel operations on *n*, *k*/*n*−bit, integer field values. However, it is only with the recent push for multimedia that the 2x to 8x speedup offered by SWAR techniques has become a concern for mainstream computing. The 1997 versions of most microprocessors incorporate hardware support for SWAR:

- AMD K6 MMX (MultiMedia eXtensions)
- Cyrix M2 MMX (MultiMedia eXtensions)
- Digital Alpha MAX (MultimediA eXtensions)
- Hewlett−Packard PA−RISC MAX (Multimedia Acceleration eXtensions)
- Intel Pentium II & Pentium with MMX (MultiMedia eXtensions)
- Microunity Mediaprocessor SIGD (Single Instruction on Groups of Data)
- MIPS Digital Media eXtension (MDMX, pronounced Mad Max)
- Sun SPARC V9 VIS (Visual Instruction Set)

There are a few holes in the hardware support provided by the new microprocessors, quirks like only supporting some operations for some field sizes. It is important to remember, however, that you don't need any hardware support for many SWAR operations to be efficient. For example, bitwise operations are not affected by the logical partitioning of a register.

4.1 SWAR: What Is It Good For?

Although *every* modern processor is capable of executing with at least some SWAR parallelism, the sad fact is that even the best SWAR−enhanced instruction sets do not support very general−purpose parallelism. In fact, many people have noticed that the performance difference between Pentium and "Pentium with MMX technology" is often due to things like the larger L1 cache that coincided with appearance of MMX. So, realistically, what is SWAR (or MMX) good for?

- Integers only, the smaller the better. Two 32−bit values fit in a 64−bit MMX register, but so do eight one−byte characters or even an entire chess board worth of one−bit values. Note: there *will be a floating−point version of MMX*, although very little has been said about it at this writing. Cyrix has posted a set of slides, ftp://ftp.cyrix.com/developr/mpf97rm.pdf, that includes a few comments about **MMFP**. Apparently, MMFP will support two 32−bit floating−point numbers to be packed into a 64−bit MMX register; combining this with two MMFP pipelines will yield four single−precision FLOPs per clock.
- SIMD or vector−style parallelism. The same operation is applied to all fields simultaneously. There are ways to nullify the effects on selected fields (i.e., equivalent to SIMD enable masking), but they complicate coding and hurt performance.
- Localized, regular (preferably packed), memory reference patterns. SWAR in general, and MMX in particular, are terrible at randomly−ordered accesses; gathering a vector `x[y]` (where `y` is an index array) is prohibitively expensive.

These are serious restrictions, but this type of parallelism occurs in many parallel algorithms − not just multimedia applications. For the right type of algorithm, SWAR is more effective than SMP or cluster parallelism... and it doesn't cost anything to use it.

4.2 Introduction To SWAR Programming

The basic concept of SWAR, SIMD Within A Register, is that operations on word−length registers can be used to speed−up computations by performing SIMD parallel operations on *nk/n*−bit field values. However, making use of SWAR technology can be awkward, and some SWAR operations are actually more expensive than the corresponding sequences of serial operations because they require additional instructions to enforce the field partitioning.

To illustrate this point, let's consider a greatly simplified SWAR mechanism that manages four 8−bit fields within each 32−bit register. The values in two registers might be represented as:

```
         PE3     PE2     PE1     PE0
      +-------+-------+-------+-------+
Reg0  | D 7:0 | C 7:0 | B 7:0 | A 7:0 |
      +-------+-------+-------+-------+
Reg1  | H 7:0 | G 7:0 | F 7:0 | E 7:0 |
      +-------+-------+-------+-------+
```

This simply indicates that each register is viewed as essentially a vector of four independent 8−bit integer values. Alternatively, think of `A` and `E` as values in Reg0 and Reg1 of processing element 0 (PE0), `B` and `F` as values in PE1's registers, and so forth.

The remainder of this document briefly reviews the basic classes of SIMD parallel operations on these integer vectors and how these functions can be implemented.

Polymorphic Operations

Some SWAR operations can be performed trivially using ordinary 32−bit integer operations, without concern for the fact that the operation is really intended to operate independently in parallel on these 8−bit fields. We call any such SWAR operation *polymorphic*, since the function is unaffected by the field types (sizes).

Testing if any field is non−zero is polymorphic, as are all bitwise logic operations. For example, an ordinary bitwise−and operation (C's `&` operator) performs a bitwise and no matter what the field sizes are. A simple bitwise and of the above registers yields:

```
          PE3       PE2       PE1       PE0
      +---------+---------+---------+---------+
Reg2  | D&H 7:0 | C&G 7:0 | B&F 7:0 | A&E 7:0 |
      +---------+---------+---------+---------+
```

Because the bitwise and operation always has the value of result bit *k* affected only by the values of the operand bit *k* values, all field sizes are supported using the same single instruction.

Partitioned Operations

Unfortunately, lots of important SWAR operations are not polymorphic. Arithmetic operations such as add, subtract, multiply, and divide are all subject to carry/borrow interactions between fields. We call such SWAR operations *partitioned*, because each such operation must effectively partition the operands and result to prevent interactions between fields. However, there are actually three different methods that can be used to achieve this effect.

Partitioned Instructions

Perhaps the most obvious approach to implementing partitioned operations is to provide hardware support for "partitioned parallel instructions" that cut the carry/borrow logic between fields. This approach can yield the highest performance, but it requires a change to the processor's instruction set and generally places many restrictions on field size (e.g., 8−bit fields might be supported, but not 12−bit fields).

The AMD/Cyrix/Intel MMX, Digital MAX, HP MAX, and Sun VIS all implement restricted versions of partitioned instructions. Unfortunately, these different instruction set extensions have significantly different restrictions, making algorithms somewhat non−portable between them. For example, consider the following sampling of partitioned operations:

Instruction	AMD/Cyrix/Intel MMX	DEC MAX	HP MAX	Sun VIS
Absolute Difference		8		8
Merge Maximum		8, 16		
Compare	8, 16, 32			16, 32
Multiply	16			8x16
Add	8, 16, 32		16	16, 32

In the table, the numbers indicate the field sizes, in bits, for which each operation is supported. Even though the table omits many instructions including all the more exotic ones, it is clear that there are many differences. The direct result is that high−level languages (HLLs) really are not very effective as programming models, and portability is generally poor.

Unpartitioned Operations With Correction Code

Implementing partitioned operations using partitioned instructions can certainly be efficient, but what do you do if the partitioned operation you need is not supported by the hardware? The answer is that you use a series of ordinary instructions to perform the operation with carry/borrow across fields, and then correct for the undesired field interactions.

This is a purely software approach, and the corrections do introduce overhead, but it works with fully general field partitioning. This approach is also fully general in that it can be used either to fill gaps in the hardware support for partitioned instructions, or it can be used to provide full functionality for target machines that have no hardware support at all. In fact, by expressing the code sequences in a language like C, this approach allows SWAR programs to be fully portable.

The question immediately arises: precisely how inefficient is it to simulate SWAR partitioned operations using unpartitioned operations with correction code? Well, that is certainly the $64k question... but many operations are not as difficult as one might expect.

Consider implementing a four−element 8−bit integer vector add of two source vectors, `x+y`, using ordinary 32−bit operations.

An ordinary 32−bit add might actually yield the correct result, but not if any 8−bit field carries into the next field. Thus, our goal is simply to ensure that such a carry does not occur. Because adding two *k*−bit fields generates an at most *k*+1 bit result, we can ensure that no carry occurs by simply "masking out" the most significant bit of each field. This is done by bitwise anding each operand with `0x7f7f7f7f` and then performing an ordinary 32−bit add.

```
t = ((x & 0x7f7f7f7f) + (y & 0x7f7f7f7f));
```

That result is correct... except for the most significant bit within each field. Computing the correct value for each field is simply a matter of doing two 1−bit partitioned adds of the most significant bits from `x` and `y` to the 7−bit carry result which was computed for `t`. Fortunately, a 1−bit partitioned add is implemented by an ordinary exclusive or operation. Thus, the result is simply:

```
(t ^ ((x ^ y) & 0x80808080))
```

Ok, well, maybe that isn't so simple. After all, it is six operations to do just four adds. However, notice that the number of operations is not a function of how many fields there are... so, with more fields, we get speedup. In fact, we may get speedup anyway simply because the fields were loaded and stored in a single (integer vector) operation, register availability may be improved, and there are fewer dynamic code scheduling dependencies (because partial word references are avoided).

Controlling Field Values

While the other two approaches to partitioned operation implementation both center on getting the maximum space utilization for the registers, it can be computationally more efficient to instead control the field values so that inter−field carry/borrow events should never occur. For example, if we know that all the field values being added are such that no field overflow will occur, a partitioned add operation can be implemented using an ordinary add instruction; in fact, given this constraint, an ordinary add instruction appears polymorphic, and is usable for any field sizes without correction code. The question thus becomes how to ensure that field values will not cause carry/borrow events.

One way to ensure this property is to implement partitioned instructions that can restrict the range of field values. The Digital MAX vector minimum and maximum instructions can be viewed as hardware support for clipping field values to avoid inter−field carry/borrow.

However, suppose that we do not have partitioned instructions that can efficiently restrict the range of field values... is there a sufficient condition that can be cheaply imposed to ensure carry/borrow events will not interfere with adjacent fields? The answer lies in analysis of the arithmetic properties. Adding two *k*−bit numbers generates a result with at most *k*+1 bits; thus, a field of *k*+1 bits can safely contain such an operation despite using ordinary instructions.

Thus, suppose that the 8−bit fields in our earlier example are now 7−bit fields with 1−bit "carry/borrow spacers":

```
              PE3          PE2          PE1          PE0
      +----+-------+----+-------+----+-------+----+-------+
Reg0  | D' | D 6:0 | C' | C 6:0 | B' | B 6:0 | A' | A 6:0 |
      +----+-------+----+-------+----+-------+----+-------+
```

A vector of 7−bit adds is performed as follows. Let us assume that, prior to the start of any partitioned operation, all the carry spacer bits (`A'`, `B'`, `C'`, and `D'`) have the value 0. By simply executing an ordinary add operation, all the fields obtain the correct 7−bit values; however, some spacer bit values might now be 1. We can correct this by just one more conventional operation, masking−out the spacer bits. Our 7−bit integer vector add, `x+y`, is thus:

```
((x + y) & 0x7f7f7f7f)
```

This is just two instructions for four adds, clearly yielding good speedup.

The sharp reader may have noticed that setting the spacer bits to 0 does not work for subtract operations. The correction is, however, remarkably simple. To compute `x−y`, we simply ensure the initial condition that the spacers in `x` are all 1, while the spacers in `y` are all 0. In the worst case, we would thus get:

```
(((x | 0x80808080) - y) & 0x7f7f7f7f)
```

However, the additional bitwise or operation can often be optimized out by ensuring that the operation generating the value for `x` used `| 0x80808080` rather than `& 0x7f7f7f7f` as the last step.

Which method should be used for SWAR partitioned operations? The answer is simply "whichever yields the best speedup." Interestingly, the ideal method to use may be different for different field sizes within the same program running on the same machine.

Communication & Type Conversion Operations

Although some parallel computations, including many operations on image pixels, have the property that the *i*th value in a vector is a function only of values that appear in the *i*th position of the operand vectors, this is generally not the case. For example, even pixel operations such as smoothing require values from adjacent pixels as operands, and transformations like FFTs require more complex (less localized) communication patterns.

It is not difficult to efficiently implement 1−dimensional nearest neighbor communication for SWAR using unpartitioned shift operations. For example, to move a value from `PE`*i* to `PE`(*i*+1), a simple shift operation suffices. If the fields are 8−bits in length, we would use:

```
(x << 8)
```

Still, it isn't always quite that simple. For example, to move a value from `PE`*i* to `PE`(*i*−1), a simple shift operation might suffice... but the C language does not specify if shifts right preserve the sign bit, and some machines only provide signed shift right. Thus, in the general case, we must explicitly zero the potentially replicated sign bits:

```
((x >> 8) & 0x00ffffff)
```

Adding "wrap−around connections" is also reasonably efficient using unpartitioned shifts. For example, to move a value from `PE`*i* to `PE`(*i*+1) with wraparound:

```
((x << 8) | ((x >> 24) & 0x000000ff))
```

The real problem comes when more general communication patterns must be implemented. Only the HP MAX instruction set supports arbitrary rearrangement of fields with a single instruction, which is called `Permute`. This `Permute` instruction is really misnamed; not only can it perform an arbitrary permutation of the fields, but it also allows repetition. In short, it implements an arbitrary `x[y]` operation.

Unfortunately, `x[y]` is very difficult to implement without such an instruction. The code sequence is generally both long and inefficient; in fact, it is sequential code. This is very disappointing. The relatively high speed of `x[y]` operations in the MasPar MP1/MP2 and Thinking Machines CM1/CM2/CM200 SIMD supercomputers was one of the key reasons these machines performed well. However, `x[y]` has always been slower than nearest neighbor communication, even on those supercomputers, so many algorithms have been designed to minimize the need for `x[y]` operations. In short, without hardware support, it is probably best to develop SWAR algorithms as though `x[y]` wasn't legal... or at least isn't cheap.

Recurrence Operations (Reductions, Scans, etc.)

A recurrence is a computation in which there is an apparently sequential relationship between values being computed. However, if these recurrences involve associative operations, it may be possible to recode the computation using a tree−structured parallel algorithm.

The most common type of parallelizable recurrence is probably the class known as associative reductions. For example, to compute the sum of a vector's values, one commonly writes purely sequential C code like:

```
t = 0;
for (i=0; i<MAX; ++i) t += x[i];
```

However, the order of the additions is rarely important. Floating point and saturation math can yield different answers if the order of additions is changed, but ordinary wrap−around integer additions will yield the same results independent of addition order. Thus, we can re−write this sequence into a tree−structured parallel summation in which we first add pairs of values, then pairs of those partial sums, and so forth, until a single final sum results. For a vector of four 8−bit values, just two addition steps are needed; the first step does two 8−bit adds, yielding two 16−bit result fields (each containing a 9−bit result):

```
t = ((x & 0x00ff00ff) + ((x >> 8) & 0x00ff00ff));
```

The second step adds these two 9−bit values in 16−bit fields to produce a single 10−bit result:

```
((t + (t >> 16)) & 0x000003ff)
```

Actually, the second step performs two 16−bit field adds... but the top 16−bit add is meaningless, which is why the result is masked to a single 10−bit result value.

Scans, also known as "parallel prefix" operations, are somewhat harder to implement efficiently. This is because, unlike reductions, scans produce partitioned results. For this reason, scans can be implemented using a fairly obvious sequence of partitioned operations.

4.3 MMX SWAR Under Linux

For Linux, IA32 processors are our primary concern. The good news is that AMD, Cyrix, and Intel all implement the same MMX instructions. However, MMX performance varies; for example, the K6 has only one MMX pipeline − the Pentium with MMX has two. The only really bad news is that Intel is still running those stupid MMX commercials.... ;−)

There are really three approaches to using MMX for SWAR:

1. Use routines from an MMX library. In particular, Intel has developed several "performance libraries," http://developer.intel.com/drg/tools/ad.htm, that offer a variety of hand−optimized routines for common multimedia tasks. With a little effort, many non−multimedia algorithms can be reworked to enable some of the most compute−intensive portions to be implemented using one or more of these library routines. These libraries are not currently available for Linux, but could be ported.
2. Use MMX instructions directly. This is somewhat complicated by two facts. The first problem is that MMX might not be available on the processor, so an alternative implementation must also be provided. The second problem is that the IA32 assembler generally used under Linux does not

currently recognize MMX instructions.

3. Use a high−level language or module compiler that can directly generate appropriate MMX instructions. Such tools are currently under development, but none is yet fully functional under Linux. For example, at Purdue University (http://dynamo.ecn.purdue.edu/~hankd/SWAR/) we are currently developing a compiler that will take functions written in an explicitly parallel C dialect and will generate SWAR modules that are callable as C functions, yet make use of whatever SWAR support is available, including MMX. The first prototype module compilers were built in Fall 1996, however, bringing this technology to a usable state is taking much longer than was originally expected.

In summary, MMX SWAR is still awkward to use. However, with a little extra effort, the second approach given above can be used now. Here are the basics:

1. You cannot use MMX if your processor does not support it. The following GCC code can be used to test if MMX is supported on your processor. It returns 0 if not, non−zero if it is supported.

```
inline extern
int mmx_init(void)
{
        int mmx_available;

        __asm__ __volatile__ (
                /* Get CPU version information */
                "movl $1, %%eax\n\t"
                "cpuid\n\t"
                "andl $0x800000, %%edx\n\t"
                "movl %%edx, %0"
                : "=q" (mmx_available)
                : /* no input */
        );
        return mmx_available;
}
```

2. An MMX register essentially holds one of what GCC would call an `unsigned long long`. Thus, memory−based variables of this type become the communication mechanism between your MMX modules and the C programs that call them. Alternatively, you can declare your MMX data as any 64−bit aligned data structure (it is convenient to ensure 64−bit alignment by declaring your data type as a `union` with an `unsigned long long` field).
3. If MMX is available, you can write your MMX code using the `.byte` assembler directive to encode each instruction. This is painful stuff to do by hand, but not difficult for a compiler to generate. For example, the MMX instruction `PADDB MM0,MM1` could be encoded as the GCC in−line assembly code:

```
__asm__ __volatile__ (".byte 0x0f, 0xfc, 0xc1\n\t");
```

   Remember that MMX uses some of the same hardware that is used for floating point operations, so code intermixed with MMX code must not invoke any floating point operations. The floating point stack also should be empty before executing any MMX code; the floating point stack is normally empty at the beginning of a C function that does not use floating point.
4. Exit your MMX code by executing the `EMMS` instruction, which can be encoded as:

```
__asm__ __volatile__ (".byte 0x0f, 0x77\n\t");
```

If the above looks very awkward and crude, it is. However, MMX is still quite young.... future versions of this document will offer better ways to program MMX SWAR.

5.Linux−Hosted Attached Processors

Although this approach has recently fallen out of favor, it is virtually impossible for other parallel processing methods to achieve the low cost and high performance possible by using a Linux system to host an attached parallel computing system. The problem is that very little software support is available; you are pretty much on your own.

5.1 A Linux PC Is A Good Host

In general, attached parallel processors tend to be specialized to perform specific types of functions.

Before becoming discouraged by the fact that you are somewhat on your own, it is useful to understand that, although it may be difficult to get a Linux PC to appropriately host a particular system, a Linux PC is one of the few platforms well suited to this type of use.

PCs make a good host for two primary reasons. The first is the cheap and easy expansion capability; resources such as more memory, disks, networks, etc., are trivially added to a PC. The second is the ease of interfacing. Not only are ISA and PCI bus prototyping cards widely available, but the parallel port offers reasonable performance in a completely non−invasive interface. The IA32 separate I/O space also facilitates interfacing by providing hardware I/O address protection at the level of individual I/O port addresses.

Linux also makes a good host OS. The free availability of full source code, and extensive "hacking" guides, obviously are a tremendous help. However, Linux also provides good near−real−time scheduling, and there is even a true real−time version of Linux at http://luz.cs.nmt.edu/~rtlinux/. Perhaps even more important is the fact that while providing a full UNIX environment, Linux can support development tools that were written to run under Microsoft DOS and/or Windows. MSDOS programs can execute within a Linux process using `dosemu` to provide a protected virtual machine that can literally run MSDOS. Linux support for Windows 3.xx programs is even more direct: free software such as `wine`, http://www.linpro.no/wine/, simulates Windows 3.11 well enough for most programs to execute correctly and efficiently within a UNIX/X environment.

The following two sections give examples of attached parallel systems that I'd like to see supported under Linux....

5.2 Did You DSP That?

There is a thriving market for high−performance DSP (Digital Signal Processing) processors. Although these chips were generally designed to be embedded in application−specific systems, they also make great attached parallel computers. Why?

- Many of them, such as the Texas Instruments (http://www.ti.com/) TMS320 and the Analog Devices (http://www.analog.com/) SHARC DSP families, are designed to construct parallel machines with little or no "glue" logic.
- They are cheap, especially per MIP or MFLOP. Including the cost of basic support logic, it is not unheard of for a DSP processor to be one tenth the cost of a PC processor with comparable performance.
- They do not use much power nor generate much heat. This means that it is possible to have a bunch of these chips powered by a conventional PC's power supply − and enclosing them in your PC's case will not turn it into an oven.
- There are strange−looking things in most DSP instruction sets that high−level (e.g., C) compilers are unlikely to use well − for example, "Bit Reverse Addressing." Using an attached parallel system, it is possible to straightforwardly compile and run most code on the host, while running the most time−consuming few algorithms on the DSPs as carefully hand−tuned code.
- These DSP processors are not really designed to run a UNIX−like OS, and generally are not very good as stand−alone general−purpose computer processors. For example, many do not have memory management hardware. In other words, they work best when hosted by a more general−purpose machine... such as a Linux PC.

Although some audio cards and modems include DSP processors that Linux drivers can access, the big payoff comes from using an attached parallel system that has four or more DSP processors.

Because the Texas Instruments TMS320 series, http://www.ti.com/sc/docs/dsps/dsphome.htm, has been very popular for a long time, and it is trivial to construct a TMS320−based parallel processor, there are quite a few such systems available. There are both integer−only and floating−point capable versions of the TMS320; older designs used a somewhat unusual single−precision floating−point format, but the new models support IEEE formats. The older TMS320C4x (aka, 'C4x) achieves up to 80 MFLOPS using the TI−specific single−precision floating−point format; in contrast, a single 'C67x will provide up to 1 GFLOPS single−precision or 420 MFLOPS double−precision for IEEE floating point calculations, using a VLIW−based chip architecture called VelociTI. Not only is it easy to configure a group of these chips as a multiprocessor, but in a single chip, the 'C8x multiprocessor will provide a 100 MFLOPS IEEE floating−point RISC master processor along with either two or four integer slave DSPs.

The other DSP processor family that has been used in more than a few attached parallel systems lately is the SHARC (aka, ADSP−2106x) from Analog Devices http://www.analog.com/. These chips can be configured as a 6−processor shared memory multiprocessor without external glue logic, and larger systems also can be configured using six 4−bit links/chip. Most of the larger systems seem targeted to military applications, and are a bit pricey. However, Integrated Computing Engines, Inc., http://www.iced.com/, makes an interesting little two−board PCI card set called GreenICE. This unit contains an array of 16 SHARC processors, and is capable of delivering a peak speed of about 1.9 GFLOPS using a single−precision IEEE format. GreenICE costs less than $5,000.

In my opinion, attached parallel DSPs really deserve a lot more attention from the Linux parallel processing community....

5.3 FPGAs And Reconfigurable Logic Computing

If parallel processing is all about getting the highest speedup, then why not build custom hardware? Well, we all know the answers; it costs too much, takes too long to develop, becomes useless when we change the algorithm even slightly, etc. However, recent advances in electrically reprogrammable FPGAs (Field Programmable Gate Arrays) have nullified most of those objections. Now, the gate density is high enough so that an entire simple processor can be built within a single FPGA, and the time to reconfigure (reprogram) an FPGA has also been dropping to a level where it is reasonable to reconfigure even when moving from one phase of an algorithm to the next.

This stuff is not for the weak of heart: you'll have to work with hardware description languages like VHDL for the FPGA configuration, as well as writing low−level code to interface to programs on the Linux host system. However, the cost of FPGAs is low, and especially for algorithms operating on low−precision integer data (actually, a small superset of the stuff SWAR is good at), FPGAs can perform complex operations just about as fast as you can feed them data. For example, simple FPGA−based systems have yielded better−than−supercomputer times for searching gene databases.

There are other companies making appropriate FPGA−based hardware, but the following two companies represent a good sample.

Virtual Computer Company offers a variety of products using dynamically reconfigurable SRAM−based Xilinx FPGAs. Their 8/16 bit "Virtual ISA Proto Board" http://www.vcc.com/products/isa.html is less than $2,000.

The Altera ARC−PCI (Altera Reconfigurable Computer, PCI bus), http://www.altera.com/html/new/pressrel/pr_arc−pci.html, is a similar type of card, but uses Altera FPGAs and a PCI bus interface rather than ISA.

Many of the design tools, hardware description languages, compilers, routers, mappers, etc., come as object code only that runs under Windows and/or DOS. You could simply keep a disk partition with DOS/Windows on your host PC and reboot whenever you need to use them, however, many of these software packages may work under Linux using `dosemu` or Windows emulators like `wine`.

6.Of General Interest

The material covered in this section applies to all four parallel processing models for Linux.

6.1 Programming Languages And Compilers

I am primarily known as a compiler researcher, so I'd like to be able to say that there are lots of really great compilers automatically generating efficient parallel code for Linux systems. Unfortunately, the truth is that it is hard to beat the performance obtained by expressing your parallel program using various explicit communication and other parallel operations within C code that is compiled by GCC.

The following language/compiler projects represent some of the best efforts toward producing reasonably efficient code from high−level languages. Generally, each is reasonably effective for the kinds of programming tasks it targets, but none is the powerful general−purpose language and compiler system that will make you forever stop writing C programs to compile with GCC... which is fine. Use these languages and compilers as they were intended, and you'll be rewarded with shorter development times, easier debugging and maintenance, etc.

There are plenty of languages and compilers beyond those listed here (in alphabetical order). A list of freely available compilers (most of which have nothing to do with Linux parallel processing) is at http://www.idiom.com/free−compilers/.

Fortran 66/77/PCF/90/HPF/95

At least in the scientific computing community, there will always be Fortran. Of course, now Fortran doesn't mean the same thing it did in the 1966 ANSI standard. Basically, Fortran 66 was pretty simple stuff. Fortran 77 added tons of features, the most noticeable of which were the improved support for character data and the change of `DO` loop semantics. PCF (Parallel Computing Forum) Fortran attempted to add a variety of parallel processing support features to 77. Fortran 90 is a fully−featured modern language, essentially adding C++−like object−oriented programming features and parallel array syntax to the 77 language. HPF (High−Performance Fortran, http://www.crpc.rice.edu/HPFF/home.html), which has itself gone through two versions (HPF−1 and HPF−2), is essentially the enhanced, standardized, version of what many of us used to know as CM Fortran, MasPar Fortran, or Fortran D; it extends Fortran 90 with a variety of parallel processing enhancements, largely focussed on specifying data layouts. Finally, Fortran 95 represents a relatively minor enhancement and refinement of 90.

What works with C generally can also work with `f2c`, `g77` (a nice Linux−specific overview is at http://linux.uni−regensburg.de/psi_linux/gcc/html_g77/g77_91.html), or the commercial Fortran 90/95 products from http://extweb.nag.co.uk/nagware/NCNJNKNM.html. This is because all of these compilers eventually come down to the same code−generation used in the back−end of GCC.

Commercial Fortran parallelizers that can generate code for SMPs are available from http://www.kai.com/ and http://www.psrv.com/vast/vast_parallel.html. It is not clear if these compilers will work for SMP Linux, but it should be possible given that the standard POSIX threads (i.e., LinuxThreads) work under SMP Linux.

The Portland Group, http://www.pgroup.com/, has commercial parallelizing HPF Fortran (and C, C++) compilers that generate code for SMP Linux; they also have a version targeting clusters using MPI or PVM. FORGE/spf/xHPF products at http://www.apri.com/ might also be useful for SMPs or clusters.

Freely available parallelizing Fortrans that might be made to work with parallel Linux systems include:

- ADAPTOR (Automatic DAta Parallelism TranslaTOR, http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html), which can translate HPF into Fortran 77/90 code with MPI or PVM calls, but does not mention Linux.
- Fx http://www.cs.cmu.edu/~fx/Fx at Carnegie Mellon targets some workstation clusters, but Linux?
- HPFC (prototype HPF Compiler, http://www.cri.ensmp.fr/~coelho/hpfc.html) generates Fortran 77 code with PVM calls. Is it usable on a Linux cluster?
- Can PARADIGM (PARAllelizing compiler for DIstributed−memory General−purpose Multicomputers, http://www.crhc.uiuc.edu/Paradigm/) be used with Linux?
- The Polaris compiler, http://ece.www.ecn.purdue.edu/~eigenman/polaris/, generates Fortran code for shared memory multiprocessors, and may soon be retargeted to PAPERS Linux clusters.
- PREPARE, http://www.irisa.fr/EXTERNE/projet/pampa/PREPARE/prepare.html, targets MPI clusters... it is not clear if it can generate code to run on IA32 processors.
- Combining ADAPT and ADLIB, shpf (Subset High Performance Fortran compilation system, http://www.ccg.ecs.soton.ac.uk/Projects/shpf/shpf.html) is public domain and generates Fortran 90 with MPI calls... so, if you have a Fortran 90 compiler under Linux....
- SUIF (Stanford University Intermediate Form, see http://suif.stanford.edu/) has parallelizing compilers for both C and Fortran. This is also the focus of the National Compiler Infrastructure Project... so, is anybody targeting parallel Linux systems?

I'm sure that I have omitted many potentially useful compilers for various dialects of Fortran, but there are so many that it is difficult to keep track. In the future, I would prefer to list only those compilers known to work with Linux. Please email comments and/or corrections to pplinux@ecn.purdue.edu.

GLU (Granular Lucid)

GLU (Granular Lucid) is a very high−level programming system based on a hybrid programming model that combines intensional (Lucid) and imperative models. It supports both PVM and TCP sockets. Does it run under Linux? More information is available at http://www.csl.sri.com/GLU.html.

Jade And SAM

Jade is a parallel programming language that extends C to exploit coarse−grain concurrency in sequential, imperative programs. It assumes a distributed shared memory model, which is implemented by SAM for workstation clusters using PVM. More information is available at http://suif.stanford.edu/~scales/sam.html.

Mentat And Legion

Mentat is an object−oriented parallel processing system that works with workstation clusters and has been ported to Linux. Mentat Programming Language (MPL) is an object−oriented programming language based on C++. The Mentat run−time system uses something vaguely resembling non−blocking remote procedure calls. More information is available at http://www.cs.virginia.edu/~mentat/.

Legion http://www.cs.virginia.edu/~legion/ is built on top on Mentat, providing the appearance of a single virtual machine across wide−area networked machines.

MPL (MasPar Programming Language)

Not to be confussed with Mentat's MPL, this language was originally developed as the native parallel C dialect for the MasPar SIMD supercomputers. Well, MasPar isn't really in that business any more (they are now NeoVista Solutions, http://www.neovista.com, a data mining company), but their MPL compiler was built using GCC, so it is still freely available. In a joint effort between the University of Alabama at Huntsville and Purdue University, MasPar's MPL has been retargeted to generate C code with AFAPI calls (see section 3.6), and thus runs on both Linux SMPs and clusters. The compiler is, however, somewhat buggy... see http://www.math.luc.edu/~laufer/mspls/papers/cohen.ps.

PAMS (Parallel Application Management System)

Myrias is a company selling a software product called PAMS (Parallel Application Management System). PAMS provides very simple directives for virtual shared memory parallel processing. Networks of Linux machines are not yet supported. See http://www.myrias.com/ for more information.

Parallaxis−III

Parallaxis−III is a structured programming language that extends Modula−2 with "virtual processors and connections" for data parallelism (a SIMD model). The Parallaxis software comprises compilers for sequential and parallel computer systems, a debugger (extensions to the gdb and xgbd debugger), and a large variety of sample algorithms from different areas, especially image processing. This runs on sequential Linux systems... an old version supported various parallel targets, and the new version also will (e.g., targeting a PVM cluster). More information is available at http://www.informatik.uni−stuttgart.de/ipvr/bv/p3/p3.html.

pC++/Sage++

pC++/Sage++ is a language extension to C++ that permits data−parallel style operations using "collections of objects" from some base "element" class. It is a preprocessor generating C++ code that can run under PVM. Does it run under Linux? More information is available at http://www.extreme.indiana.edu/sage/.

SR (Synchronizing Resources)

SR (Synchronizing Resources) is a concurrent programming language in which resources encapsulate processes and the variables they share; operations provide the primary mechanism for process interaction. SR provides a novel integration of the mechanisms for invoking and servicing operations. Consequently, all of local and remote procedure call, rendezvous, message passing, dynamic process creation, multicast, and semaphores are supported. SR also supports shared global variables and operations.

It has been ported to Linux, but it isn't clear what parallelism it can execute with. More information is available at http://www.cs.arizona.edu/sr/www/index.html.

ZPL And IronMan

ZPL is an array−based programming language intended to support engineering and scientific applications. It generates calls to a simple message−passing interface called IronMan, and the few functions which constitute this interface can be easily implemented using nearly any message−passing system. However, it is primarily targeted to PVM and MPI on workstation clusters, and Linux is supported. More information is available at http://www.cs.washington.edu/research/projects/orca3/zpl/www/.

6.2 Performance Issues

There are a lot of people who spend a lot of time benchmarking particular motherboards, network cards, etc., trying to determine which is the best. The problem with that approach is that by the time you've been able to benchmark something, it is no longer the best available; it even may have been taken off the market and replaced by a revised model with entirely different properties.

Buying PC hardware is like buying orange juice. Usually, it is made with pretty good stuff no matter what company name is on the label. Few people know, or care, where the components (or orange juice concentrate) came from. That said, there are some hardware differences that you should pay attention to. My advice is simply that you be aware of what you can expect from the hardware under Linux, and then focus your attention on getting rapid delivery, a good price, and a reasonable policy for returns.

An excellent overview of the different PC processors is given in http://www.pcguide.com/ref/cpu/fam/; in fact, the whole WWW site http://www.pcguide.com/ is full of good technical overviews of PC hardware. It is also useful to know a bit about performance of specific hardware configurations, and the Linux Benchmarking HOWTO http://sunsite.unc.edu/LDP/HOWTO/Benchmarking−HOWTO.html is a good place to start.

The Intel IA32 processors have many special registers that can be used to measure the performance of a running system in exquisite detail. Intel VTune, http://developer.intel.com/design/perftool/vtune/, uses the performance registers extensively in a very complete code−tuning system... that unfortunately doesn't run under Linux. A loadable module device driver, and library routines, for accessing the Pentium performance registers is available from http://www.cs.umd.edu/users/akinlar/driver.html. Keep in mind that these performance registers are different on different IA32 processors; this code works only with Pentium, not with 486, Pentium Pro, Pentium II, K6, etc.

Another comment on performance is appropriate, especially for those of you who want to build big clusters and put them in small spaces. At least some modern processors incorporate thermal sensors and circuits that are used to slow the internal clock rate if operating temperature gets too high (an attempt to reduce heat output and improve reliability). I'm not suggesting that everyone should go buy a peltier device (heat pump) to cool each CPU, but you should be aware that high operating temperature does not just shorten component life − it also can directly reduce system performance. Do not arrange your computers in physical configurations that block airflow, trap heat within confined areas, etc.

Finally, performance isn't just speed, but also reliability and availability. High reliability means that your system almost never crashes, even when components fail... which generally requires special features like redundant power supplies and hot−swap motherboards. That usually isn't cheap. High availability refers to the concept that your system is available for use nearly all the time... the system may crash when components fail, but the system is quickly repaired and rebooted. There is a High−Availability HOWTO that discusses many of the basic issues. However, especially for clusters, high availablity can be achieved simply by having a few spares. I recommend at least one spare, and prefer to have at least one spare for every 16 machines in a large cluster. Discarding faulty hardware and replacing it with a spare can yield both higher availability and lower cost than a maintenance contract.

6.3 Conclusion − It's Out There

So, is anybody doing parallel processing using Linux? Yes!

It wasn't very long ago that a lot of people were wondering if the death of many parallel−processing supercomputer companies meant that parallel processing was on its way out. I didn't think it was dead then (see http://dynamo.ecn.purdue.edu/~hankd/Opinions/pardead.html for a fun overview of what I think really happened), and it seems quite clear now that parallel processing is again on the rise. Even Intel, which just recently stopped making parallel supercomputers, is proud of the parallel processing support in things like MMX and the upcoming IA64 EPIC (Explicitly Parallel Instruction Computer).

If you search for "Linux" and "parallel" with your favorite search engine, you'll find quite a few places are involved in parallel processing using Linux. In particular, Linux PC clusters seem to be popping−up everywhere. The appropriateness of Linux, combined with the low cost and high performance of PC hardware, have made parallel processing using Linux a popular approach to supercomputing for both small, budget−constrained, groups and large, well−funded, national research laboratories.

Various projects listed elsewhere in this document maintain lists of "kindred" research sites that have similar parallel Linux configurations. However, at http://yara.ecn.purdue.edu/~pplinux/Sites/, there is a hypertext document intended to provide photographs, descriptions, and contact information for all the various sites using Linux systems for parallel processing. To have information about your site posted there:

- You must have a "permanent" parallel Linux site: an SMP, cluster of machines, SWAR system, or PC with attached processor, which is configured to allow users to *execute parallel programs under Linux*. A Linux−based software environment (e.g., PVM, MPI, AFAPI) that directly supports parallel processing must be installed on the system. However, the hardware need not be dedicated to parallel processing under Linux, and may be used for completely different purposes when parallel programs are not being run.
- Request that your site be listed. Send your site information to pplinux@ecn.purdue.edu. Please follow the format used in other entries for your site information. *No site will be listed without an explicit request from the contact person for that site.*

There are 14 clusters in the current listing, but we are aware of at least several dozen Linux clusters world−wide. Of course, listing does not imply any endorsement, etc.; our hope is simply to increase awareness, research, and collaboration involving parallel processing using Linux.