Next Previous Contents

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.linux.org.uk/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 hankd@engr.uky.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 hankd@engr.uky.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://aggregate.org/LDP/, 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.

Next Previous Contents