Beowulf HOWTO

Table of Contents

Table of Contents

Beowulf HOWTO

Jacek Radajewski and Douglas Eadline

v1.1.1, 22 November 1998

This document introduces the Beowulf Supercomputer architecture and provides background information on parallel programming, including links to other more specific documents, and web pages.

1.Preamble

2.Introduction

3.Architecture Overview

4.System Design

5.Beowulf Resources

6.Source code

1.Preamble

1.1 Disclaimer

We will not accept any responsibility for any incorrect information within this document, nor for any damage it might cause when applied.

1.2 Copyright

1.3 About this HOWTO

Jacek Radajewski started work on this document in November 1997 and was soon joined by Douglas Eadline. Over a few months the Beowulf HOWTO grew into a large document, and in August 1998 it was split into three documents: Beowulf HOWTO, Beowulf Architecture Design HOWTO, and the Beowulf Installation and Administration HOWTO. Version 1.0.0 of the Beowulf HOWTO was released to the Linux Documentation Project on 11 November 1998. We hope that this is only the beginning of what will become a complete Beowulf Documentation Project.

1.4 About the authors

- Jacek Radajewski works as a Network Manager, and is studying for an honors degree in computer science at the University of Southern Queensland, Australia. Jacek's first contact with Linux was in 1995 and it was love at first sight. Jacek built his first Beowulf cluster in May 1997 and has been playing with the technology ever since, always trying to find new and better ways of setting things up. You can contact Jacek by sending e−mail to jacek@usq.edu.au
- Douglas Eadline, Ph.D. is President and Principal Scientist at Paralogic, Inc., Bethlehem, PA, USA. Trained as Physical/Analytical Chemist, he has been involved with computers since 1978 when he built his first single board computer for use with chemical instrumentation. Dr. Eadline's interests now include Linux, Beowulf clusters, and parallel algorithms. Dr. Eadline can be contacted by sending email to deadline@plogic.com

1.5 Acknowledgements

The writing of the Beowulf HOWTO was a long proces and is finally complete, thanks to many individuals. I would like to thank the following people for their help and contribution to this HOWTO.

- Becky for her love, support, and understanding.
- Tom Sterling, Don Becker, and other people at NASA who started the Beowulf project.
- Thanh Tran−Cong and the Faculty of Engineering and Surveying for making the *topcat* Beowulf machine available for experiments.
- My supervisor Christopher Vance for many great ideas.
- My friend Russell Waldron for great programming ideas, his general interest in the project, and support.
- My friend David Smith for proof reading this document.
- Many other people on the Beowulf mailing list who provided me with feedback and ideas.
- All the people who are responsible for the Linux operating system and all the other free software packages used on *topcat* and other Beowulf machines.

2.Introduction

As the performance of commodity computer and network hardware increase, and their prices decrease, it becomes more and more practical to build parallel computational systems from off−the−shelf components, rather than buying CPU time on very expensive Supercomputers. In fact, the price per performance ratio of a Beowulf type machine is between three to ten times better than that for traditional supercomputers. Beowulf architecture scales well, it is easy to construct and you only pay for the hardware as most of the software is free.

2.1 Who should read this HOWTO ?

This HOWTO is designed for a person with at least some exposure to the Linux operating system. Knowledge of Beowulf technology or understanding of more complex operating system and networking concepts is not essential, but some exposure to parallel computing would be advantageous (after all you must have some reason to read this document). This HOWTO will not answer all possible questions you might have about Beowulf, but hopefully will give you ideas and guide you in the right direction. The purpose of this HOWTO is to provide background information, links and references to more advanced documents.

2.2 What is a Beowulf ?

Famed was this Beowulf: far flew the boast of him, son of Scyld, in the Scandian lands. So becomes it a youth to quit him well with his father's friends, by fee and gift, that to aid him, aged, in after days, come warriors willing, should war draw nigh, liegemen loyal: by lauded deeds shall an earl have honor in every clan. Beowulf is the earliest surviving epic poem written in English. It is a story about a hero of great strength and courage who defeted a monster called Grendel. See History to find out more about the Beowulf hero.

There are probably as many Beowulf definitions as there are people who build or use Beowulf Supercomputer facilities. Some claim that one can call their system Beowulf only if it is built in the same way as the NASA's original machine. Others go to the other extreme and call Beowulf any system of workstations running parallel code. My definition of Beowulf fits somewhere between the two views described above, and is based on many postings to the Beowulf mailing list:

Beowulf is a multi computer architecture which can be used for parallel computations. It is a system which usually consists of one server node, and one or more client nodes connected together via Ethernet or some other network. It is a system built using commodity hardware components, like any PC capable of running Linux, standard Ethernet adapters, and switches. It does not contain any custom hardware components and is trivially reproducible. Beowulf also uses commodity software like the Linux operating system, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). The server node controls the whole cluster and serves files to the client nodes. It is also the cluster's console and gateway to the outside world. Large Beowulf machines might have more than one server node, and possibly other nodes dedicated to particular tasks, for example consoles or monitoring stations. In most cases client nodes in a Beowulf system are dumb, the dumber the better. Nodes are configured and controlled by the server node, and do only what they are told to do. In a disk−less client configuration, client nodes don't even know their IP address or name until the server tells them what it is. One of the main differences between Beowulf and a Cluster of Workstations (COW) is the fact that Beowulf behaves more like a single machine rather than many workstations. In most cases client nodes do not have keyboards or monitors, and are accessed only via remote login or possibly serial terminal. Beowulf nodes can be thought of as a CPU + memory package which can be plugged in to the cluster, just like a CPU or memory module can be plugged into a motherboard.

Beowulf is not a special software package, new network topology or the latest kernel hack. Beowulf is a technology of clustering Linux computers to form a parallel, virtual supercomputer. Although there are many software packages such as kernel modifications, PVM and MPI libraries, and configuration tools which make the Beowulf architecture faster, easier to configure, and much more usable, one can build a Beowulf class machine using standard Linux distribution without any additional software. If you have two networked Linux computers which share at least the `/home` file system via NFS, and trust each other to execute remote shells (rsh), then it could be argued that you have a simple, two node Beowulf machine.

2.3 Classification

Beowulf systems have been constructed from a variety of parts. For the sake of performance some non−commodity components (i.e. produced by a single manufacturer) have been employed. In order to account for the different types of systems and to make discussions about machines a bit easier, we propose the following simple classification scheme:

CLASS I BEOWULF:

This class of machines built entirely from commodity "off−the−shelf" parts. We shall use the "Computer Shopper" certification test to define commodity "off−the−shelf" parts. (Computer Shopper is a 1 inch thick monthly magazine/catalog of PC systems and components.) The test is as follows:

A CLASS I Beowulf is a machine that can be assembled from parts found in at least 3 nationally/globally circulated advertising catalogs.

The advantages of a CLASS I system are:

- hardware is available form multiple sources (low prices, easy maintenance)
- no reliance on a single hardware vendor
- driver support from Linux commodity
- usually based on standards (SCSI, Ethernet, etc.)

The disadvantages of a CLASS I system are:

- best performance may require CLASS II hardware

CLASS II BEOWULF

A CLASS II Beowulf is simply any machine that does not pass the Computer Shopper certification test. This is not a bad thing. Indeed, it is merely a classification of the machine.

The advantages of a CLASS II system are:

- Performance can be quite good!

The disadvantages of a CLASS II system are:

- driver support may vary
- reliance on single hardware vendor
- may be more expensive than CLASS I systems.

One CLASS is not necessarily better than the other. It all depends on your needs and budget. This classification system is only intended to make discussions about Beowulf systems a bit more succinct. The "System Design" section may help determine what kind of system is best suited for your needs.

3.Architecture Overview

3.1 What does it look like ?

I think that the best way of describing the Beowulf supercomputer architecture is to use an example which is very similar to the actual Beowulf, but familiar to most system administrators. The example that is closest to a Beowulf machine is a Unix computer laboratory with a server and a number of clients. To be more specific I'll use the DEC Alpha undergraduate computer laboratory at the Faculty of Sciences, USQ as the example. The server computer is called *beldin* and the client machines are called *scilab01*, *scilab02*, *scilab03*, up to *scilab20*. All clients have a local copy of the Digital Unix 4.0 operating system installed, but get the user file space (`/home`) and `/usr/local` from the server via NFS (Network File System). Each client has an entry for the server and all the other clients in its `/etc/hosts.equiv` file, so all clients can execute a remote shell (rsh) to all others. The server machine is a NIS server for the whole laboratory, so account information is the same across all the machines. A person can sit at the *scilab02* console, login, and have the same environment as if he logged onto the server or *scilab15*. The reason all the clients have the same look and feel is that the operating system is installed and configured in the same way on all machines, and both the user's `/home` and `/usr/local` areas are physically on the server and accessed by the clients via NFS. For more information on NIS and NFS please read the NIS and NFS HOWTOs.

3.2 How to utilise the other nodes ?

Now that we have some idea about the system architecture, let us take a look at how we can utilise the available CPU cycles of the machines in the computer laboratory. Any person can logon to any of the machines, and run a program in their home directory, but they can also spawn the same job on a different machine simply by executing remote shell. For example, assume that we want to calculate the sum of the square roots of all integers between 1 and 10 inclusive. We write a simple program called `sigmasqrt` (please see source code) which does exactly that. To calculate the sum of the square roots of numbers from 1 to 10 we execute :

```
[jacek@beldin sigmasqrt]$ time ./sigmasqrt 1 10
22.468278

real    0m0.029s
user    0m0.001s
sys     0m0.024s
```

The `time` command allows us to check the wall−clock (the elapsed time) of running this job. As we can see, this example took only a small fraction of a second (0.029 sec) to execute, but what if I want to add the square root of integers from 1 to 1 000 000 000 ? Let us try this, and again calculate the wall−clock time.

```
[jacek@beldin sigmasqrt]$ time ./sigmasqrt 1 1000000000
21081851083600.559000
```

```
real    16m45.937s
user    16m43.527s
sys     0m0.108s
```

This time, the execution time of the program is considerably longer. The obvious question to ask is what can we do to speed up the execution time of the job? How can we change the way the job is running to minimize the wall−clock time of running this job? The obvious answer is to split the job into a number of sub−jobs and to run these sub−jobs in parallel on all computers. We could split one big addition task into 20 parts, calculating one range of square roots and adding them on each node. When all nodes finish the calculation and return their results, the 20 numbers could be added together to obtain the final solution. Before we run this job we will make a named pipe which will be used by all processes to write their results.

```
[jacek@beldin sigmasqrt]$ mkfifo output
[jacek@beldin sigmasqrt]$ ./prun.sh & time cat output | ./sum
[1] 5085
21081851083600.941000
[1]+  Done                    ./prun.sh

real    0m58.539s
user    0m0.061s
sys     0m0.206s
```

This time we get about 58.5 seconds. This is the time from starting the job until all the nodes have finished their computations and written their results into the pipe. The time does not include the final addition of the twenty numbers, but this time is a very small fraction of a second and can be ignored. We can see that there is a significant improvement in running this job in parallel. In fact the parallel job ran about 17 times faster, which is very reasonable for a 20 fold increase in the number of CPUs. The purpose of the above example is to illustrate the simplest method of parallelising concurrent code. In practice such simple examples are rare and different techniques (PVM and PMI APIs) are used to achieve the parallelism.

3.3 How does Beowulf differ from a COW ?

The computer laboratory described above is a perfect example of a Cluster of Workstations (COW). So what is so special about Beowulf, and how is it different from a COW? The truth is that there is not much difference, but Beowulf does have few unique characteristics. First of all, in most cases client nodes in a Beowulf cluster do not have keyboards, mice, video cards nor monitors. All access to the client nodes is done via remote connections from the server node, dedicated console node, or a serial console. Because there is no need for client nodes to access machines outside the cluster, nor for machines outside the cluster to access client nodes directly, it is a common practice for the client nodes to use private IP addresses like the 10.0.0.0/8 or 192.168.0.0/16 address ranges (RFC 1918 http://www.alternic.net/rfcs/1900/rfc1918.txt.html). Usually the only machine that is also connected to the outside world using a second network card is the server node. The most common ways of using the system is to access the server's console directly, or either telnet or remote login to the server node from personal workstation. Once on the server node, users can edit and compile their code, and also spawn jobs on all nodes in the cluster. In most cases COWs are used for parallel computations at night, and over weekends when people do not actually use the workstations for every day work, thus utilising idle CPU cycles. Beowulf on the other hand is a machine usually dedicated to parallel computing, and optimised for this purpose. Beowulf also gives better price/performance ratio as it is built from off−the−shelf components and runs mainly free software. Beowulf has also more single system image features which help the users to see the Beowulf cluster as a single computing workstation.

4.System Design

Before you purchase any hardware, it may be a good idea to consider the design of your system. There are basically two hardware issues involved with design of a Beowulf system: the type of nodes or computers you are going to use; and way you connect the computer nodes. There is one software issue that may effect your hardware decisions; the communication library or API. A more detailed discussion of hardware and communication software is provided later in this document.

While the number of choices is not large, there are some important design decisions that must be made when constructing a Beowulf systems. Because the science (or art) of "parallel computing" has many different interpretations, an introduction is provided below. If you do not like to read background material, you may skip this section, but it is advised that you read section Suitability before you make you final hardware decisions.

4.1 A brief background on parallel computing.

This section provides background on parallel computing concepts. It is NOT an exhaustive or complete description of parallel computing science and technology. It is a brief description of the issues that may be important to a Beowulf designer and user.

As you design and build your Beowulf, many of these issues described below will become important in your decision process. Due to its component nature, a Beowulf Supercomputer requires that we consider many factors carefully because they are now under our control. In general, it is not all that difficult to understand the issues involved with parallel computing. Indeed, once the issues are understood, your expectations will be more realistic and success will be more likely. Unlike the "sequential world" where processor speed is considered the single most important factor, processor speed in the "parallel world" is just one of several factors that will determine overall system performance and efficiency.

4.2 The methods of parallel computing

Parallel computing can take many forms. From a user's perspective, it is important to consider the advantages and disadvantages of each methodology. The following section attempts to provide some perspective on the methods of parallel computing and indicate where the Beowulf machine falls on this continuum.

Why more than one CPU?

Answering this question is important. Using 8 CPUs to run your word processor sounds a little like "over−kill" −− and it is. What about a web server, a database, a rendering program, or a project scheduler? Maybe extra CPUs would help. What about a complex simulation, a fluid dynamics code, or a data mining application. Extra CPUs definitely help in these situations. Indeed, multiple CPUs are being used to solve more and more problems.

The next question usually is: "Why do I need two or four CPUs, I will just wait for the 986 turbo−hyper chip." There are several reasons:

1. Due to the use of multi−tasking Operating Systems, it is possible to do several things at once. This is a natural "parallelism" that is easily exploited by more than one low cost CPU.
2. Processor speeds have been doubling every 18 months, but what about RAM speeds or hard disk speeds? Unfortunately, these speeds are not increasing as fast as the CPU speeds. Keep in mind most applications require "out of cache memory access" and hard disk access. Doing things in parallel is one way to get around some of these limitations.
3. Predictions indicate that processor speeds will not continue to double every 18 months after the year 2005. There are some very serious obstacles to overcome in order to maintain this trend.
4. Depending on the application, parallel computing can speed things up by any where from 2 to 500 times faster (in some cases even faster). Such performance is not available using a single processor. Even supercomputers that at one time used very fast custom processors are now built from multiple "commodity− off−the−shelf" CPUs.

If you need speed − either due to a compute bound problem and/or an I/O bound problem, parallel is worth considering. Because parallel computing is implemented in a variety of ways, solving your problem in parallel will require some very important decisions to be made. These decisions may dramatically effect portability, performance, and cost of your application.

Before we get technical, let's look take a look at a real "parallel computing problem" using an example with which we are familiar − waiting in long lines at a store.

The Parallel Computing Store

Consider a big store with 8 cash registers grouped together in the front of the store. Assume each cash register/cashier is a CPU and each customer is a computer program. The size of the computer program (amount of work) is the size of each customer's order. The following analogies can be used to illustrate parallel computing concepts.

Single−tasking Operating System

One cash register open (is in use) and must process each customer one at a time.

Computer Example: MS DOS

Multi−tasking Operating System:

One cash register open, but now we process only a part of each order at a time, move to the next person and process some of their order. Everyone "seems" to be moving through the line together, but if no one else is in the line, you will get through the line faster.

Computer Example: UNIX, NT using a single CPU

Multitasking Operating Systems with Multiple CPUs:

Now we open several cash registers in the store. Each order can be processed by a separate cash register and the line can move much faster. This is called SMP − Symmetric Multi−processing. Although there are extra cash registers open, you will still never get through the line any faster than just you and a single cash register.

Computer Example: UNIX and NT with multiple CPUs

Threads on a Multitasking Operating Systems extra CPUs

If you "break−up" the items in your order, you might be able to move through the line faster by using several cash registers at one time. First, we must assume you have a large amount of goods, because the time you invest "breaking up your order" must be regained by using multiple cash registers. In theory, you should be able to move through the line "n" times faster than before*; where "n" is the number of cash registers. When the cashiers need to get sub− totals, they can exchange information quickly by looking and talking to all the other "local" cash registers. They can even snoop around the other cash registers to find information they need to work faster. There is a limit, however, as to how many cash registers the store can effectively locate in any one place.

Amdals law will also limit the application speed−up to the slowest sequential portion of the program.

Computer Example: UNIX or NT with extra CPU on the same motherboard running multi−threaded programs.

Sending Messages on Multitasking Operating Systems with extra CPUs:

In order to improve performance, the store adds 8 cash registers at the back of the store. Because the new cash registers are far away from the front cash registers, the cashiers must call on the phone to send their sub−totals to the front of the store. This distance adds extra overhead (time) to communication between cashiers, but if communication is minimized, it is not a problem. If you have a really big order, one that requires all the cash registers, then as before your speed can be improved by using all cash registers at the same time, the extra overhead must be considered. In some cases, the store may have single cash registers (or islands of cash registers) located all over the store − each cash register (or island) must communicate by phone. Since all the cashiers working the cash registers can talk to each other by phone, it does not matter too much where they are.

Computer Example: One or several copies of UNIX or NT with extra CPUs on the same or different motherboard communicating through messages.

The above scenarios, although not exact, are a good representation of constraints placed on parallel systems. Unlike a single CPU (or cash register) communication is an issue.

4.3 Architectures for parallel computing

The common methods and architectures of parallel computing are presented below. While this description is by no means exhaustive, it is enough to understand the basic issues involved with Beowulf design.

Hardware Architectures

There are basically two ways parallel computer hardware is put together:

1. Local memory machines that communicate by messages (Beowulf Clusters)
2. Shared memory machines that communicate through memory (SMP machines)

A typical Beowulf is a collection of single CPU machines connected using fast Ethernet and is, therefore, a local memory machine. A 4 way SMP box is a shared memory machine and can be used for parallel computing − parallel applications communicate using shared memory. Just as in the computer store analogy, local memory machines (individual cash registers) can be scaled up to large numbers of CPUs, while the number of CPUs shared memory machines (the number of cash registers you can place in one spot) can have is limited due to memory contention.

It is possible, however, to connect many shared memory machines to create a "hybrid" shared memory machine. These hybrid machines "look" like a single large SMP machine to the user and are often called NUMA (non uniform memory access) machines because the global memory seen by the programmer and shared by all the CPUs can have different latencies. At some level, however, a NUMA machine must "pass messages" between local shared memory pools.

It is also possible to connect SMP machines as local memory compute nodes. Typical CLASS I motherboards have either 2 or 4 CPUs and are often used as a means to reduce the overall system cost. The Linux internal scheduler determines how these CPUs get shared. The user cannot (at this point) assign a specific task to a specific SMP processor. The user can however, start two independent processes or a threaded processes and expect to see a performance increase over a single CPU system.

Software API Architectures

There basically two ways to "express" concurrency in a program:

1. Using Messages sent between processors
2. Using operating system Threads

Other methods do exist, but these are the two most widely used. It is important to remember that the expression of concurrency is not necessary controlled by the underlying hardware. Both Messages and Threads can be implemented on SMP, NUMA−SMP, and clusters − although as explained below efficiently and portability are important issues.

Messages

Historically, messages passing technology reflected the design of early local memory parallel computers. Messages require copying data while Threads use data in place. The latency and speed at which messages can be copied are the limiting factor with message passing models. A Message is quite simple: some data and a destination processor. Common message passing APIs are PVM or MPI. Message passing can be efficiently implemented using Threads and Messages work well both on SMP machine and between clusters of machines. The advantage to using messages on an SMP machine, as opposed to Threads, is that if you decided to use clusters in the future it is easy to add machines or scale your application.

Threads

Operating system Threads were developed because shared memory SMP (symmetrical multiprocessing) designs allowed very fast shared memory communication and synchronization between concurrent parts of a program. Threads work well on SMP systems because communication is through shared memory. For this reason the user must isolate local data from global data, otherwise programs will not work properly. In contrast to messages, a large amount of copying can be eliminated with threads because the data is shared between processes (threads). Linux supports POSIX threads. The problem with threads is that it is difficult to extend them beyond one SMP machine and because data is shared between CPUs, cache coherence issues can contribute to overhead. Extending threads beyond the SMP boundary efficiently requires NUMA technology which is expensive and not natively supported by Linux. Implementing threads on top of messages has been done ((http://syntron.com/ptools/ptools_pg.htm)), but Threads are often inefficient when implemented using messages.

The following can be stated about performance:

	SMP machine performance	cluster of machines performance	scalability
messages	good	best	best
threads	best	poor*	poor*

```
* requires expensive NUMA technology.
```

Application Architecture

In order to run an application in parallel on multiple CPUs, it must be explicitly broken in to concurrent parts. A standard single CPU application will run no faster than a single CPU application on multiple processors. There are some tools and compilers that can break up programs, but parallelizing codes is not a "plug and play" operation. Depending on the application, parallelizing code can be easy, extremely difficult, or in some cases impossible due to algorithm dependencies.

Before the software issues can be addressed the concept of Suitability needs to be introduced.

4.4 Suitability

Most questions about parallel computing have the same answer:

"It all depends upon the application."

Before we jump into the issues, there is one very important distinction that needs to be made − the difference between CONCURRENT and PARALLEL. For the sake of this discussion we will define these two concepts as follows:

CONCURRENT parts of a program are those that can be computed independently.

PARALLEL parts of a program are those CONCURRENT parts that are executed on separate processing elements at the same time.

The distinction is very important, because CONCURRENCY is a property of the program and efficient PARALLELISM is a property of the machine. Ideally, PARALLEL execution should result in faster performance. The limiting factor in parallel performance is the communication speed and latency between compute nodes. (Latency also exists with threaded SMP applications due to cache coherency.) Many of the common parallel benchmarks are highly parallel and communication and latency are not the bottle neck. This type of problem can be called "obviously parallel". Other applications are not so simple and executing CONCURRENT parts of the program in PARALLEL may actually cause the program to run slower, thus offsetting any performance gains in other CONCURRENT parts of the program. In simple terms, the cost of communication time must pay for the savings in computation time, otherwise the PARALLEL execution of the CONCURRENT part is inefficient.

The task of the programmer is to determining what CONCURRENT parts of the program SHOULD be executed in PARALLEL and what parts SHOULD NOT. The answer to this will determine the EFFICIENCY of application. The following graph summarizes the situation for the programmer:

```
         | *
         | *
         | *
 % of    | *
 appli-  |  *
 cations |  *
         |  *
         |  *
         |    *
         |     *
         |      *
         |        ****
         |            ****
         |                ********************
         +-----------------------------------
          communication time/processing time
```

In a perfect parallel computer, the ratio of communication/processing would be equal and anything that is CONCURRENT could be implemented in PARALLEL. Unfortunately, Real parallel computers, including shared memory machines, are subject to the effects described in this graph. When designing a Beowulf, the user may want to keep this graph in mind because parallel efficiency depends upon ratio of communication time and processing time for A SPECIFIC PARALLEL COMPUTER. Applications may be portable between parallel computers, but there is no guarantee they will be efficient on a different platform.

IN GENERAL, THERE IS NO SUCH THING AS A PORTABLE AND EFFICIENT PARALLEL PROGRAM

There is yet another consequence to the above graph. Since efficiency depends upon the comm./process. ratio, just changing one component of the ratio does not necessary mean a specific application will perform faster. A change in processor speed, while keeping the communication speed that same may have non–intuitive effects on your program. For example, doubling or tripling the CPU speed, while keeping the communication speed the same, may now make some previously efficient PARALLEL portions of your program, more efficient if they were executed SEQUENTIALLY. That is, it may now be faster to run the previously PARALLEL parts as SEQUENTIAL. Furthermore, running inefficient parts in parallel will actually keep your application from reaching its maximum speed. Thus, by adding faster processor, you may actually slowed down your application (you are keeping the new CPU from running at its maximum speed for that application)

UPGRADING TO A FASTER CPU MAY ACTUALLY SLOW DOWN YOUR APPLICATION

So, in conclusion, to know whether or not you can use a parallel hardware environment, you need to have some insight into the suitability of a particular machine to your application. You need to look at a lot of issues including CPU speeds, compiler, message passing API, network, etc. Please note, just profiling an application, does not give the whole story. You may identify a computationally heavy portion of your program, but you do not know the communication cost for this portion. It may be that for a given system, the communication cost as do not make parallelizing this code efficient.

A final note about a common misconception. It is often stated that "a program is PARALLELIZED", but in reality only the CONCURRENT parts of the program have been located. For all the reasons given above, the program is not PARALLELIZED. Efficient PARALLELIZATION is a property of the machine.

4.5 Writing and porting parallel software

Once you decide that you need parallel computing and would like to design and build a Beowulf, a few moments considering your application with respect to the previous discussion may be a good idea.

In general there are two things you can do:

1. Go ahead and construct a CLASS I Beowulf and then "fit" your application to it. Or run existing parallel applications that you know work on your Beowulf (but beware of the portability and efficiently issues mentioned above)
2. Look at the applications you need to run on your Beowulf and make some estimations as to the type of hardware and software you need.

In either case, at some point you will need to look at the efficiency issues. In general, there are three things you need to do:

1. Determine concurrent parts of your program
2. Estimate parallel efficiently
3. Describing the concurrent parts of your program

Let's look at these one at a time.

Determine concurrent parts of your program

This step is often considered "parallelizing your program". Parallelization decisions will be made in step 2. In this step, you need to determine data dependencies.

>From a practical standpoint, applications may exhibit two types of concurrency: compute (number crunching) and I/O (database). Although in many cases compute and I/O concurrency are orthogonal, there are application that require both. There are tools available that can perform concurrency analysis on existing applications. Most of these tools are designed for FORTRAN. There are two reasons FORTRAN is used: historically most number crunching applications were written in FORTRAN and it is easier to analyze. If no tools are available, then this step can be some what difficult for existing applications.

Estimate parallel efficiency

Without the help of tools, this step may require trial and error tests or just a plain old educated guess. If you have a specific application in mind, try to determine if it is CPU limited (compute bound) or hard disk limited (I/O bound). The requirements of your Beowulf may be quite different depending upon your needs. For example, a compute bound problem may need a few very fast CPUs and high speed low latency network, while an I/O bound problem may work better with more slower CPUs and fast Ethernet.

This recommendation often comes as a surprise to most people because, the standard assumption is that faster processor are always better. While this is true if your have an unlimited budget, real systems may have cost constraints that should be maximized. For I/O bound problems, there is a little known rule (called the Eadline−Dedkov Law) that is quite helpful:

For two given parallel computers with the same cumulative CPU performance index, the one which has slower processors (and a probably correspondingly slower interprocessor communication network) will have better performance for I/O−dominant applications.

While the proof of this rule is beyond the scope of this document, you find it interesting to download the paper *Performance Considerations for I/O−Dominant Applications on Parallel Computers* (Postscript format 109K) (ftp://www.plogic.com/pub/papers/exs−pap6.ps)

Once you have determined what type of concurrency you have in your application, you will need to estimate how efficient it will be in parallel. See Section Software for a description of Software tools.

In the absence of tools, you may try to guess your way through this step. If a compute bound loop measured in minutes and the data can be transferred in seconds, then it might be a good candidate for parallelization. But remember, if you take a 16 minute loop and break it into 32 parts, and your data transfers require several seconds per part, then things are going to get tight. You will reach a point of diminishing returns.

Describing the concurrent parts of your program

There are several ways to describe concurrent parts of your program:

1. Explicit parallel execution
2. Implicit parallel execution

The major difference between the two is that explicit parallelism is determined by the user where implicit parallelism is determined by the compiler.

Explicit Methods

These are basically method where the user must modify source code specifically for a parallel computer. The user must either add messages using PVM or MPI or add threads using POSIX threads. (Keep in mind however, threads can not move between SMP motherboards).

Explicit methods tend to be the most difficult to implement and debug. Users typically embed explicit function calls in standard FORTRAN 77 or C/C++ source code. The MPI library has added some functions to make some standard parallel methods easier to implement (i.e. scatter/gather functions). In addition, it is also possible to use standard libraries that have been written for parallel computers. Keep in mind, however, the portability vs. efficiently trade−off)

For historical reasons, most number crunching codes are written in FORTRAN. For this reasons, FORTRAN has the largest amount of support (tools, libraries, etc.) for parallel computing. Many programmers now use C or re− write existing FORTRAN applications in C with the notion the C will allow faster execution. While this may be true as C is the closest thing to a universal machine code, it has some major drawbacks. The use of pointers in C makes determining data dependencies extremely difficult. Automatic analysis of pointers is extremely difficult. If you have an existing FORTRAN program and think that you might want to parallelize it in the future − DO NOT CONVERT IT TO C!

Implicit Methods

Implicit methods are those where the user gives up some (or all) of the parallelization decisions to the compiler. Examples are FORTRAN 90, High Performance FORTRAN (HPF), Bulk Synchronous Parallel (BSP), and a whole collection of other methods that are under development.

Implicit methods require the user to provide some information about the concurrent nature of their application, but the compiler will then make many decisions about how to execute this concurrency in parallel. These methods provide some level of portability and efficiency, but there is still no "best way" to describe a concurrent problem for a parallel computer.

5.Beowulf Resources

5.1 Starting Points

- Beowulf mailing list. To subscribe send mail to beowulf−request@cesdis.gsfc.nasa.gov with the word *subscribe* in the message body.
- Beowulf Homepage http://www.beowulf.org
- Extreme Linux http://www.extremelinux.org
- Extreme Linux Software from Red Hat http://www.redhat.com/extreme

5.2 Documentation

- The latest version of the Beowulf HOWTO http://www.sci.usq.edu.au/staff/jacek/beowulf.
- Building a Beowulf System http://www.cacr.caltech.edu/beowulf/tutorial/building.html
- Jacek's Beowulf Links http://www.sci.usq.edu.au/staff/jacek/beowulf.
- Beowulf Installation and Administration HOWTO (DRAFT) http://www.sci.usq.edu.au/staff/jacek/beowulf.
- Linux Parallel Processing HOWTO http://yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto.html

5.3 Papers

- Chance Reschke, Thomas Sterling, Daniel Ridge, Daniel Savarese, Donald Becker, and Phillip Merkey *A Design Study of Alternative Network Topologies for the Beowulf Parallel Workstation*. Proceedings Fifth IEEE International Symposium on High Performance Distributed Computing, 1996. http://www.beowulf.org/papers/HPDC96/hpdc96.html
- Daniel Ridge, Donald Becker, Phillip Merkey, Thomas Sterling Becker, and Phillip Merkey. *Harnessing the Power of Parallelism in a Pile−of−PCs*. Proceedings, IEEE Aerospace, 1997. http://www.beowulf.org/papers/AA97/aa97.ps
- Thomas Sterling, Donald J. Becker, Daniel Savarese, Michael R. Berry, and Chance Res. *Achieving a Balanced Low−Cost Architecture for Mass Storage Management through Multiple Fast Ethernet Channels on the Beowulf Parallel Workstation*. Proceedings, International Parallel Processing Symposium, 1996. http://www.beowulf.org/papers/IPPS96/ipps96.html
- Donald J. Becker, Thomas Sterling, Daniel Savarese, Bruce Fryxell, Kevin Olson. *Communication Overhead for Space Science Applications on the Beowulf Parallel Workstation*. Proceedings,High Performance and Distributed Computing, 1995.

http://www.beowulf.org/papers/HPDC95/hpdc95.html
- Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, Charles V. Packer. *BEOWULF: A PARALLEL WORKSTATION FOR SCIENTIFIC COMPUTATION*. Proceedings, International Conference on Parallel Processing, 95. http://www.beowulf.org/papers/ICPP95/icpp95.html
- Papers at the Beowulf site http://www.beowulf.org/papers/papers.html

5.4 Software

- PVM − Parallel Virtual Machine http://www.epm.ornl.gov/pvm/pvm_home.html
- LAM/MPI (Local Area Multicomputer / Message Passing Interface http://www.mpi.nd.edu/lam
- BERT77 − FORTRAN conversion tool http://www.plogic.com/bert.html
- Beowulf software from Beowulf Project Page http://beowulf.gsfc.nasa.gov/software/software.html
- Jacek's Beowulf−utils ftp://ftp.sci.usq.edu.au/pub/jacek/beowulf−utils
- bWatch − cluster monitoring tool http://www.sci.usq.edu.au/staff/jacek/bWatch

5.5 Beowulf Machines

- Avalon consists of 140 Alpha processors, 36 GB of RAM, and is probably the fastest Beowulf machine, cruising at 47.7 Gflops and ranking 114th on the Top 500 list. http://swift.lanl.gov/avalon/
- Megalon−A Massively PArallel CompuTer Resource (MPACTR) consists of 14, quad CPU Pentium Pro 200 nodes, and 14 GB of RAM. http://megalon.ca.sandia.gov/description.html
- theHIVE − Highly−parallel Integrated Virtual Environment is another fast Beowulf Supercomputer. theHIVE is a 64 node, 128 CPU machine with the total of 4 GB RAM. http://newton.gsfc.nasa.gov/thehive/
- Topcat is a much smaller machine and consists of 16 CPUs and 1.2 GB RAM. http://www.sci.usq.edu.au/staff/jacek/topcat
- MAGI cluster − this is a very interesting site with many good links. http://noel.feld.cvut.cz/magi/

5.6 Other Interesting Sites

- SMP Linux http://www.linux.org.uk/SMP/title.html
- Paralogic − Buy a Beowulf http://www.plogic.com

5.7 History

- Legends − Beowulf http://legends.dm.net/beowulf/index.html
- The Adventures of Beowulf http://www.lnstar.com/literature/beowulf/beowulf.html

---

6.Source code

6.1 sum.c

```
/* Jacek Radajewski jacek@usq.edu.au */
/* 21/08/1998 */

#include <stdio.h>
#include <math.h>

int main (void) {

  double result = 0.0;
  double number = 0.0;
  char string[80];


  while (scanf("%s", string) != EOF) {

    number = atof(string);
    result = result + number;
  }

  printf("%lf\n", result);

  return 0;

}
```

6.2 sigmasqrt.c

```
/* Jacek Radajewski jacek@usq.edu.au */
/* 21/08/1998 */

#include <stdio.h>
#include <math.h>

int main (int argc, char** argv) {

  long number1, number2, counter;
  double result;
```

```
  if (argc < 3) {
    printf ("usage : %s number1 number2\n",argv[0]);
    exit(1);
  } else {
    number1 = atol (argv[1]);
    number2 = atol (argv[2]);
    result = 0.0;
  }

  for (counter = number1; counter <= number2; counter++) {
    result = result + sqrt((double)counter);
  }

  printf("%lf\n", result);

  return 0;

}
```

6.3 prun.sh

```
#!/bin/bash
# Jacek Radajewski jacek@usq.edu.au
# 21/08/1998

export SIGMASQRT=/home/staff/jacek/beowulf/HOWTO/example1/sigmasqrt

# $OUTPUT must be a named pipe
# mkfifo output

export OUTPUT=/home/staff/jacek/beowulf/HOWTO/example1/output

rsh scilab01 $SIGMASQRT         1  50000000 > $OUTPUT < /dev/null&
rsh scilab02 $SIGMASQRT  50000001 100000000 > $OUTPUT < /dev/null&
rsh scilab03 $SIGMASQRT 100000001 150000000 > $OUTPUT < /dev/null&
rsh scilab04 $SIGMASQRT 150000001 200000000 > $OUTPUT < /dev/null&
rsh scilab05 $SIGMASQRT 200000001 250000000 > $OUTPUT < /dev/null&
rsh scilab06 $SIGMASQRT 250000001 300000000 > $OUTPUT < /dev/null&
rsh scilab07 $SIGMASQRT 300000001 350000000 > $OUTPUT < /dev/null&
rsh scilab08 $SIGMASQRT 350000001 400000000 > $OUTPUT < /dev/null&
rsh scilab09 $SIGMASQRT 400000001 450000000 > $OUTPUT < /dev/null&
rsh scilab10 $SIGMASQRT 450000001 500000000 > $OUTPUT < /dev/null&
rsh scilab11 $SIGMASQRT 500000001 550000000 > $OUTPUT < /dev/null&
rsh scilab12 $SIGMASQRT 550000001 600000000 > $OUTPUT < /dev/null&
rsh scilab13 $SIGMASQRT 600000001 650000000 > $OUTPUT < /dev/null&
rsh scilab14 $SIGMASQRT 650000001 700000000 > $OUTPUT < /dev/null&
rsh scilab15 $SIGMASQRT 700000001 750000000 > $OUTPUT < /dev/null&
rsh scilab16 $SIGMASQRT 750000001 800000000 > $OUTPUT < /dev/null&
rsh scilab17 $SIGMASQRT 800000001 850000000 > $OUTPUT < /dev/null&
rsh scilab18 $SIGMASQRT 850000001 900000000 > $OUTPUT < /dev/null&
rsh scilab19 $SIGMASQRT 900000001 950000000 > $OUTPUT < /dev/null&
rsh scilab20 $SIGMASQRT 950000001 1000000000 > $OUTPUT < /dev/null&
```