Engineering a Beowulf-style Compute Cluster

Robert G. Brown
Duke University Physics Department
Durham, NC 27708-0305
rgb@phy.duke.edu

May 24, 2004

NOTE WELL: This book is in a pretty much perpetual state of being written and rewritten. This is inevitable – technology changes rapidly, I'm busy, I write and then rewrite, add and take away. So you're about to look at a *snapshot* that could be different tomorrow. Not greatly different, but different. So look at the date: May 24, 2004, and check back accordingly.

Note also that the author is not to be held legally responsible for the accuracy of the content of this book. Read the Open Publication License at the end if you have any doubts or questions about what you can and cannot do (can print and use for own purposes or distribution at the cost of media, cannot reprint and sell for a profit without my permission, and getting my permission will likely involving giving me money, that sort of thing). Use this book at your own risk.

Contents

Open Publication License

0.1 License Terms for "Engineering a Beowulf-style Compute Cluster"

0.1.1 General Terms

License is granted to copy or use this document according to the Open Public License (OPL, detailed below), which is a Public License, developed by the GNU Foundation, which applies to "open source" generic documents.

In addition there are three modifications to the OPL:

1. Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder. (This is to prevent errors from being introduced which would reflect badly on the author's professional abilities.)

2. Distribution of the work or derivative of the work in any form whatsoever that is sold for a profit is prohibited unless prior permission is obtained from the copyright holder. This is so that the copyright holder (a.k.a. the author) can make at least some money if this work is republished as a textbook or set of notes and sold commercially for – somebody's – profit. The author doesn't care about copies photocopied or locally printed and distributed free or at cost to students to support a course, except as far as the next clause is concerned.

3. The "Beverage" modification listed below applies to all non-Duke usage of this work in any form (online or in a paper publication). Note that this modification is probably not legally defensible and can be followed really pretty much according to the honor rule.

As to my personal preferences in beverages, red wine is great, beer is delightful, and Coca Cola or coffee or tea or even milk acceptable to those who for religious or personal reasons wish to avoid stressing my liver. Students and Faculty at Duke, whether in my class or not, of course, are automatically exempt from the beverage modification. It can be presumed that the fraction of their tuition that goes to pay my salary counts for any number of beverages.

0.1.2 The "Beverage" Modification to the OPL

Any user of this OPL-copyrighted material shall, upon meeting the primary author(s) of this OPL-copyrighted material for the first time under the appropriate circumstances, offer to buy him or her or them a beverage. This beverage may or may not be alcoholic, depending on the personal ethical and moral views of the offerer(s) and receiver(s). The beverage cost need not exceed one U.S. dollar (although it certainly may at the whim of the offerer:-) and may be accepted or declined with no further obligation on the part of the offerer. It is not necessary to repeat the offer after the first meeting, but it can't hurt...

0.1.3 OPEN PUBLICATION LICENSE Draft v0.4, 8 June 1999

I. REQUIREMENTS ON BOTH UNMODIFIED AND MODIFIED VERSIONS

The Open Publication works may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference (with any options elected by the author(s) and/or publisher) is displayed in the reproduction.

Proper form for an incorporation by reference is as follows:

Copyright (c) <year> by <author's name or designee>. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, vX.Y or later (the latest version is presently available at http://www.opencontent.org/openpub/).

The reference must be immediately followed with any options elected by the author(s) and/or publisher of the document (see section VI).

Commercial redistribution of Open Publication-licensed material is permitted.

Any publication in standard (paper) book form shall require the citation of the original publisher and author. The publisher and author's names shall appear on all outer surfaces of the book. On all outer surfaces of the book the original publisher's name shall be as large as the title of the work and cited as possessive with respect to the title.

II. COPYRIGHT

The copyright to each Open Publication is owned by its author(s) or designee.

III. SCOPE OF LICENSE

The following license terms apply to all Open Publication works, unless otherwise explicitly stated in the document.

Mere aggregation of Open Publication works or a portion of an Open Publication work with other works or programs on the same media shall not cause this license to apply to those other works. The aggregate work shall contain a notice specifying the inclusion of the Open Publication material and appropriate copyright notice.

SEVERABILITY. If any part of this license is found to be unenforceable in any jurisdiction, the remaining portions of the license remain in force.

NO WARRANTY. Open Publication works are licensed and provided "as is" without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement.

IV. REQUIREMENTS ON MODIFIED WORKS

All modified versions of documents covered by this license, including translations, anthologies, compilations and partial documents, must meet the following requirements:

1. The modified version must be labeled as such.
2. The person making the modifications must be identified and the modifications dated.
3. Acknowledgement of the original author and publisher if applicable must be retained according to normal academic citation practices.
4. The location of the original unmodified document must be identified.
5. The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission.

V. GOOD-PRACTICE RECOMMENDATIONS

In addition to the requirements of this license, it is requested from and strongly recommended of redistributors that:

1. If you are distributing Open Publication works on hardcopy or CD-ROM, you provide email notification to the authors of your intent to redistribute at least thirty days before your manuscript or media freeze, to give the authors time to provide updated documents. This notification should describe modifications, if any, made to the document.
2. All substantive modifications (including deletions) be either clearly marked up in the document or else described in an attachment to the document.

Finally, while it is not mandatory under this license, it is considered good form to offer a free copy of any hardcopy and CD-ROM expression of an Open Publication-licensed work to its author(s).

VI. LICENSE OPTIONS

The author(s) and/or publisher of an Open Publication-licensed document may elect certain options by appending language to the reference to or copy of the license. These options are considered part of the license instance and must be included with the license (or its incorporation by reference) in derived works.

A. To prohibit distribution of substantively modified versions without the explicit permission of the author(s). "Substantive modification" is defined as a change to the semantic content of the document, and excludes mere changes in format or typographical corrections.

To accomplish this, add the phrase 'Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.' to the license reference or copy.

B. To prohibit any publication of this work or derivative works in whole or in part in standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

To accomplish this, add the phrase 'Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.' to the license reference or copy.

OPEN PUBLICATION POLICY APPENDIX:

(This is not considered part of the license.)

Open Publication works are available in source format via the Open Publication home page at http://works.opencontent.org/.

Open Publication authors who want to include their own license on Open Publication works may do so, as long as their terms are not more restrictive than the Open Publication license.

If you have questions about the Open Publication License, please contact TBD, and/or the Open Publication Authors' List at opal@opencontent.org, via email.

Preface

0.2 Preface

To put my professional and topical acknowledgements first, where they belong: This book is dedicated to my many virtual friends on the beowulf list and elsewhere in the Linux world. I literally could not have written it without their help as there are a number of things that I write about below that I've never really done myself. I'm relying on the many reports of their experiences, good and bad, that have been added to the archives of the beowulf list over the years. I'm also (in the best "open source" tradition) relying on their feedback to correct any errors I may have made in writing this book.

Any errors that may have occurred are not their fault, of course; they are entirely mine.

I have tried in this book to create the basis for a reasonable understanding of the beowulf-class linux-based parallel (super)compute cluster. This book makes no pretence of being a text on computer science – it is intended for readers ranging from clever high school students with a few old x86 boxes and an ethernet hub to play with to senior systems programmers interested in engineering a world-class beowulf (with plenty of room in between for pointy-haired bosses, linux neophytes, hobbyists, and serious entrepreneurs). It is also deliberately light-hearted. I intend the text to be *readable and fun* as opposed to heavy and detailed.

This shouldn't seriously detract from its utility. Half of the fun (or profit) of beowulfery comes from the process of discovery where one takes the relatively simple idea of a beowulf and a few tools and crafts the best possible solution to *your* problem(s) for far less than one could purchase the solution commercially on "big iron" supercomputers. I have a small beowulf at home, and so can you (for as little as a two or three thousand dollars). I also have a *larger* beowulf at work (the Duke University Physics Department), and so can you (for a current cost that ranges between perhaps $500 and $8500 per brand new node, depending on just what and how much of it you get).

There is good reason to get involved in the beowulf game. One day beowulfs will play the *best* virtual reality games, allow you or your kids to make movies like "Toy Story" in the privacy of your own home (with a toolbox of predefined objects and characters so all you have to do is provide an upper-level script), solve some of the most puzzling problems of the universe, model chemicals in rational drug design packages, simulate nuclear explosions to allow advanced weapons to be designed without testing, permit fabulous optimizations to be performed with advanced genetic algorithms, and...

Well, actually beowulfs or beowulf-like architectures are *already* doing most of these things one place or another, and far more besides. The beowulf design is one of the best designs for doing parallel work because of three things: It is an *extensible*, *scalable* design built out of *cheap commodity parts* (where the word "cheap" has to be understood as "compared to the alternatives"). For that reason beowulf-style cluster computing (as a phenomenon) has grown from a handful of places and people five to ten years ago to literally cover the earth today. Its growth continues unabated today, driven by one of the strongest of human urges[1].

At this point, there are beowulfs or beowulf-like clusters in place in universities and government research centers on all the continents of the world. In many cases, the beowulf route is the *only* way these universities or research centers can acquire the computational power they need at an affordable cost. If I can build a beowulf at home, so can the physics students and faculty in a physics department in Venezuela, or Thailand, and for about the same cost. It is one way, as my friends overseas have pointed out in off-list conversations, that entrepreneurs and businesses in small countries can compete with even the biggest companies with the biggest computational facilities in the United States, in spite of ruinous and misguided export restrictions that prevent the international sale of most high powered computer systems except to carefully selected (in all the senses of the word) countries and facilities. Yes, beowulfs are also about freedom and opportunity.

Up to now, beowulfs have been relatively rare in the corporate environment, at least in the United States. It is not that corporations are unaware of the power of linux-based cluster computer environments – a lot of the web-farms in use today harness this power with equal benefits in terms of low cost and scalability. It's just that (as the book should make clear) parallelizing a big program is a complex task, which inhibits the development of commercial parallel applications.

However, in my ever-so-humble opinion, this period is about to come to an end. This is for several reasons. First of all, the beowulf design has shaken down into a "recipe" that will work for a wide range of applications. This recipe is extremely cheap and simple to implement, which allows software developers to actually design software for a "known" target architecture. In the past this has been impossible with many beowulfish clusters being "one of a kind" designs.

Second, advances on the hardware front, especially in the related realms of modular computers and high speed networks, promise a new generation of mass-market-commodity off-the-shelf components that can be assembled into "parallel supercomputers" with just the right software glue. It should come as no surprise that linux is a prime choice for the operating system to run on a lot of those small systems, since it has the right kind of modular architecture to run well with limited resources and "grow" as resources are added or connected. Is it any surprise that Linus Torvalds (the central and original inventor/creator of linux, which by now is being written and advanced by a small army of the world's best systems programmers) is working for Transmeta, a company that makes processors for ultralight mobile network-ready computers? I think not.

[1] I'm speaking, of course, of the urge to *shop* (and get the best bargain for your money). You probably thought I was talking about *sex* or *survival* or something like that, didn't you.

Transmeta is also not the only company working on small, fast, modular computer units.

Similarly, considerable energy is being expended working on advanced networks that may eventually form the communications channels for such a modular approach to computing. Really significant advances are a bit slower here (as the problems to overcome are not easy to solve) but networking has *already* reached the critical cost/benefit point to enable "recipe" beowulfs to be built for very little money. The eight port 100BT switch I use in my beowulf at home cost me a bit over $200 six months ago. Now it can be purchased for less than $150 – networking nodes together can be accomplished for as little as $40-50/node.

It is strictly downhill (in price/performance) from here. The next few years will see ever cheaper, ever faster network devices accompanied by new *kinds* of system interconnection devices that in a few years will permit nodes to be added at speeds ten or more times that of today for that *same* $40-50/node. On the other hand, the higher speed processors cost a *lot* more to build as their speed is cranked up, and the effect of the higher speed is less and less visible to ordinary computer users.

Beowulfery represents a different (and less expensive) way to achieve and surpass the same speed. I believe that this will lead to a fundamental redesign of the personal computer. Just as computers now have an expansion bus that allows various peripheral devices to be attached, future "computers" may well have an expansion bus that allows additional "compute nodes" to be attached – computers themselves may be mini-beowulfs that run software developed using the principles discussed in this book. This sort of thing has actually been tried a number of times in the past, but the systems in question have lacked proper software support (especially at the operating system level) and the appropriate hardware support as well. The next time the idea resurfaces, though, this will likely not be the case. Watch for it on your IPO screens.

Even if this particular vision fails to come to pass, though, the future of the beowulf-style (super)compute cluster is assured. As the book should make clear, there is simply no way to do any better than a beowulf design for many, many kinds of tasks (where better means better in price/peformance, not raw performance at any price). If you have a problem that involves a *lot* of computation and don't have a lot of money, it's the only game in town.

To conclude with my more mundane acknowledgements, I'd also like to thank my lovely wife, Susan Isbey and my three boys Patrick, William and Sam for their patience with my supercomputing "hobby". It isn't every home where the boys are forbidden to reboot a system for three days because it is running a calculation. Also, I can sometimes be a bit cranky after staying up all night writing or working (something I do quite a lot), and they've had to live with this. Thanks, guys.

Part I

Introduction and Overview

Chapter 1

Introduction

The Tao of Cluster Computing

In a little known book of ancient wisdom appears the following Koan:

> *The Devil finds work*
> *For idle systems*
> *Nature abhors a NoOp*

The sages have argued about the meaning of this for centicycles, some contending that idle systems are easily turned to evil tasks, others arguing that whoever uses an idle system must be possessed of the Devil and should be smote with a sucker rod until purified.

I myself interpret "Devil" to be an obvious mistranslation of the word "Daemon". It is for this reason, my son, that I wish to place a simple daemon on your system so that Nature is satisfied, for it is clear that a NoOp is merely a Void waiting to be filled...

This is the true Tao.

–rgb

The following might be considered a "recipe" for a beowulf-class cluster supercomputer:

1. Buy a pile of M^{2}COTS (Mass Market Commodity-Off-The-Shelf) PC's for "nodes". Details (graphics adapter or no, processor speed and family, amount of memory, UP or SMP, presence and size of disk) unimportant, as long as they "work" in the configuration purchased.

2. Add a nice, cheap 100BT Network Interface Card (NIC) to each. Connect each NIC to nice, cheap 100BT switch to interconnect all nodes[1].

[1]Hmmm. If you are reading this footnote, it's possible that you didn't understand that I

3. Add Linux and various "ExtremeLinux/Beowulf" packages to support distributed parallel computing; PVM, MPI, MOSIX, maybe more[2].

4. Blow your code away by running it in parallel...

Before I discuss the "recipe" further, there are some technical things that differentiate various sorts of cluster computing setups[3]. Many arrangements that more or less conform to the recipe above are not really beowulfs but are rather NOWs (network of workstations) or COWs (cluster of workstations) or POPs (Pile of PC's). Furthermore, some perfectly legitimate beowulfishly[4] architected clusters are used to provide failover protection or high availability (e.g. webserver farms, transaction processing clusters) are not really beowulfs.

Clearly, we need to specify in some detail the answer to two questions, the first a FAQ on the beowulf list (sometimes answered when it isn't even asked). The first is "What's a Beowulf". The second is "What is this book going to discuss". So let's get to it.

meant "100 BaseT Fast Ethernet Switch" or what that is. And I don't have the time or space to tell you right here. Check out the Beowulf Hardware appendix.

[2]Which are similarly described in the "Beowulf Software" appendix.

[3]I'm trying to write an *entertaining* book as much as a useful one (if I can't please you one way, maybe I can please you the other, eh). I'm therefore going to write in a "folksy" first person instead of an "academic" (and dry) third person. I'm also going to insert all sorts of parenthetical comments (like this one) as parenthetical comments or footnotes. I can only hope that this doesn't make you run screaming from the room after reading for five minutes.

[4]You have to love English. Take a noun (beowulf). Adjectivize it (beowulfish). Then adverbize the resulting adjective (beowulfishly). Nuttin' to it. Not to mention the fact that any noun can be verbed.

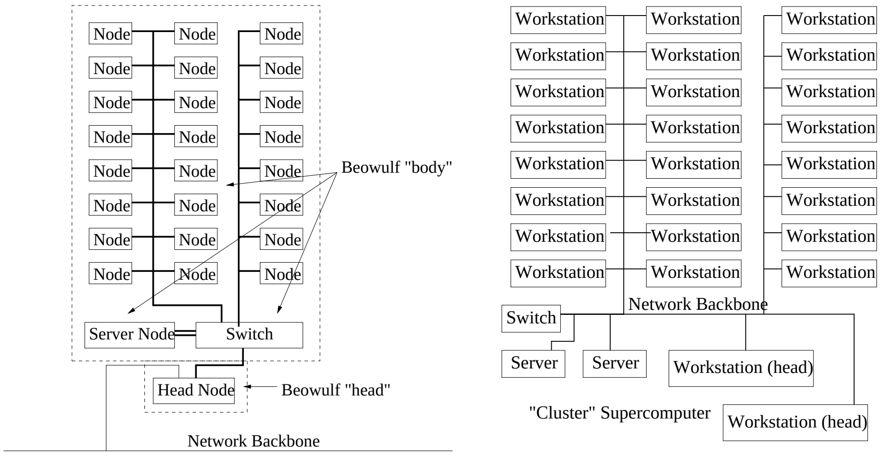


Figure 1.1: Beowulf (left) vs Mixed Cluster – NOW,COW (right)

1.1 What's a Beowulf?

Examine the schema in figure 1.1. On the left, we have the "true beowulf". The accepted definition of a true beowulf is that it is a cluster of computers ("compute nodes") interconnected with a network with the following characteristics:

- The nodes are dedicated to the beowulf and serve no other purpose.
- The network(s) on which the nodes reside is(are) dedicated to the beowulf and serve(s) no other purpose.
- The nodes are M^{2}COTS computers. An essential part of the beowulf definition (that distinguishes it from, for example a vendor-produced massively parallel processor – MPP – system) is that its compute nodes are mass produced commodities, readily available "off the shelf", and hence relatively inexpensive.
- The network is also a COTS entity (if not actually "mass market" – some beowulf networks are sold pretty much only to beowulf builders), at least to the extent that it must integrate with M^{2}COTS computers and hence must interconnect through a standard (e.g. PCI) bus. Again, this is primarily to differentiate it from vendor-produced MPP systems where the network and CPUs are custom-integrated at very high cost.
- The nodes all run *open source software.*
- The resulting cluster is used for *High Performance Computing* (HPC, also called "parallel supercomputing" and other names).

These are the *necessary* requirements. Note well that they *both* specify certain key features of the layout and architecture *and* contain an application specification. A cluster "like" a beowulf cluster that fails in any of these requirements may be many things, but a beowulf is not one of them. In addition there are a few other things that characterize *most* beowulfs and that at least some would argue belong in the "necessary" list above.

- The nodes all run a variant of *Linux* as their base operating system. Linux/Gnu and the Beowulf project and the associated Extreme Linux project have a long and entangled history. The original beowulf was built upon Linux and Gnu – indeed most of the ethernet device drivers originally provided with Linux were developed by Don Becker for the Beowulf project.

- There is one "special" node generally called the "head" or "master" node. This node often has a monitor and keyboard and has network connections *both* on the "inside" network (connecting it to the rest of the "slave" nodes) *and* on the "outside" to a general purpose organizational internetwork. This node often act as a server for e.g. shared disk space.

- Generally the nodes are all *identical* – configured with the same CPU, motherboard, network, memory, disk(s) and so forth and are neatly racked up or stacked up in shelves in a single room in a spatially contiguous way.

- Generally the nodes are all running just *one* calculation at a time (or none).

These requirements are a bit "softer" – a purist (and the list *does have* its purists) might insist on one or more of them but at some point a list of criteria like this obstructs creativity and development as much as it provides a "proven" structure. As you can probably tell, I'm not a purist, although I appreciate the virtues of the primary definition and hope to communicate a full understanding of those virtues in the following chapters. Let's examine these secondary requirements and make editorial comments.

Many folks on the beowulf list (including myself) are perfectly happy to recognize a cluster with all the architectural characteristics but running FreeBSD (an open source, unix-like operating system) as a "beowulf", while they would (and periodically do, rather "enthusiastically") reject the same cluster running closed-source Windows NT.

The point of beowulfery has never been to glorify Linux per se (however much or little it might deserve glorification) but rather to explore the possibilities of building supercomputers out of the mass-market electronic equivalent of coat hangers and chewing gum. An *essential* part of this is an open source operating system and development environment, as one's problems and optimizations will often require modification of e.g. the kernel, the network device drivers, or other task-specific optimizations that require the sources. Let's understand this.

Debugging certain problems has historically *required* access to the kernel sources and has been none too easy even with them in hand. If a network parallel computation fails, is it because of your parallel code or because the network driver or TCP stack is broken? If you use a closed-source, proprietary operating system and try to seek vendor support for a problem like this, the uniform answer they'll give you will be "your parallel program" even though your code works fine on somebody *else's* operating system and many problems encountered in beowulfery have indeed turned out to be in the kernel. Sometimes those problems are *features* of the kernel or its components.

Deliberate features of the TCP stack, for example, are intended to produce robustness across a wide area network but can cause nasty failures of your TCP-based parallel codes[5]. If you encounter one of these problems, you personally may not be able to fix it, but because Linux is open source, it is fairly likely that one of the real kernel-hacking deities of the beowulf or kernel list will be able to help you. Sometimes "hacks" of the kernel that "break" it (or mistune it) for general purpose usage can be just the ticket to improve efficiency or resolve some beowulf-specific bottleneck.

Indeed, several important beowulf tools have required a custom kernel to operate, and the only reason that these tools exist at all is because a group of "visionaries" who also happened to be damn good programmers had access to the kernel sources and could graft in the modifications essential to their tool or service. MOSIX is one example. bproc and the Scyld beowulf package are another[6].

Access to the (open, non-proprietary kernel and other) source is thus *essential* to beowulfery, but it doesn't (in my opinion) *have* to be Linux/GNU source. A warning for newbies, though – if you want to be able to get "maximum help" from the list, it's a good idea to stick with Linux because it is what most of the list members use and you'll get the most and best help that way. For example, Don Becker, one of the primary inventors of the beowulf, goes to Linux meetings and conferences and Expos, not (as far as I know) FreeBSD conferences and Expos. So do most of the other "experts" on the beowulf list. There are persons on the list running FreeBSD-based beowulfs (as well as a number of general Unix gurus who don't much care what flavor of *nix you run – they can manage it), but not too many.

The second requirement originates from the notion that a beowulf is architecturally a "single parallel supercomputer" dedicated only to supercomputing and hence has a single point of presence on an outside network, generally named something interesting and evocative[7]. However, there are sometimes virtues in having more than one head node and are *often* virtues in having a separate disk server node or even a whole farm of "disk nodes"[8]. Beowulf design is best driven

[5] This has actually happened. See e.g. http://www.icase.edu/coral/LinuxTCP2.html for details.

[6] All of these are discussed later. Be patient.

[7] The list of names of "registered" beowulfs on the beowulf website contains entries like "Grendel" (Clemson), "Loki" (Los Alamos), "Brahma" (Duke), "Medusa" (New Mexico State), and "Valhalla" (University of Missouri) as well as more whimsical names such as "Wonderland" (University of Texas at Austin).

[8] See for example the Parallel Virtual File System (PVFS) being developed at Clemson. This effort promises to parallelize disk access *in* a beowulf as an integrated part of the beowulf's

(and extended) by one's needs of the moment and vision of the future and not by a mindless attempt to slavishly follow the original technical definition anyway.

The third requirement is also associated with the idea that the supercomputer is a single entity and hence ought to live in just one place, but has a more practical basis. One reason for the isolation and dedication is to enable "fine grained synchronous" parallel calculations (code with "barriers" – which will be defined and discussed later) where one needs to be able to predict fairly accurately how long a node will take to complete a given parallel step of the calculation. This is much easier if all the nodes are identical. Otherwise one has to construct some sort of table of their differential speeds (in memory-size dependent context, which can vary considerably, see the chapter on hardware profiling and microbenchmarking) and write your program to distribute itself in such a way that all nodes still complete steps approximately synchronously.

However, it is easy to *start* with all nodes identical if one is "rich" and/or buys them all at once, but difficult to *keep* them that way as nodes (or parts of nodes) break and are replaced, or as Moore's Law inexorably advances system performance and you want or need to buy new nodes to upgrade your beowulf[9]. Again, it isn't worth quibbling about whether mixing 800 MHz nodes with 400 MHz nodes (while otherwise preserving the beowulf architecture) makes the resulting system "not a beowulf". This is especially true if one's task partitions in some way that permits both newer and older nodes to be efficiently used. The PVFS project[10] is just one kind of task partitioning that would do so – run the PVFS on the older nodes (where speed and balance are likely determined by the disk and network and not memory or CPU anyway) and run the calculation on the newer nodes.

One of the major philosophical motivations for beowulfery has always been to *save money* – to get *more* computing (and get more of Your Favorite Work done) for *less* money. That's why you bought this book[11], remember? If you didn't care about money you would have called up one of the many Big Iron companies who would have been more than happy to relieve you of a million or four dollars for what a few hundred thousand might buy you in a beowulfish architecture and never had to learn or do a damn thing[12]. Mixed node architectures extends

parallel operations. Would I call a beowulf-like collection of compute nodes mixed with disk nodes (and possibly other kinds of specialized nodes) a beowulf? I would.

[9]Moore's Law will be discussed later. It is an empirical observation that at any given price point computer performance has doubled approximately every 9-12 months for the last forty or more years. No kidding. This means that a 16 node beowulf that you're very proud of initially can be replaced by a *single node* within three or so years. Moore's Law makes *any* node or beowulf or computer design a thing of purely transient beauty.

[10]You are reading *all* the footnotes, aren't you? No? Sigh. Why did I bother to write them, then? Go back and read the last few...

[11]If, in fact, you bought this book at all. The truly cheap will be reading it a painful page at a time off the website instead of investing the truly trivial amount of money at their favorite computer bookstore necessary to ensure that I actually make a royalty and that they can read in bed. Once I actually get it published so you CAN buy it, of course.

[12]Except to pay, of course. Over and over again. Repeatedly. Big Iron supercomputing is *expensive*. Best of all, after you've owned your million-plus dollar supercomputer for as few as four or five years, you can typically sell it for as much as $3500 on the open market – they

the useful lifetime of the nodes you buy and *can* ultimately save you money.

Similar considerations attend the use of a single beowulf to run more than one different computation at a time, which is an obvious win in the event that calculation A saturates its parallel speedup when it is using only half the nodes. If one can (and it would seem silly not to) then one could always run calculation A on some of the 800 MHz nodes and calculation B (which might be less fine grained or could even be embarrassingly parallel) on the rest and all the older 400 MHz nodes without the cluster suddenly hopping out of the "true beowulf" category.

So now you know what a beowulf is. What about something that looks a lot like a beowulf (matches on "most" of the necessary criteria, but misses on just one or two counts)? It's not a beowulf, but it might still behave like one and be similarly *useful* and *cost effective*. Sometimes so cost effective that it is literally *free* – a secondary use of hardware you already have lying around or in use for other purposes.

This brings us to the second question (and second figure). The "cluster supercomputer" schematized there looks a lot like a beowulf. There are a bunch of commodity workstations that can function as "nodes" in a parallel calculation. There is a commodity network whereby they can be controlled and communicate with one another as the calculation proceeds. There are server(s). There are "head nodes" (which are themselves just workstations). We can certainly imagine that this cluster is running Linux and is being used to perform HPC calculations *among other things*.

Such a "cluster" is called variously a "Network of Workstations" (NOW), a "Cluster of Workstations" (COW), a "Pile of PC's" (POP) or an "Ordinary Linked Departmental" (OLD) network[13].

A cluster like this isn't, technically a beowulf. It isn't isolated. The "nodes" are often doing several things at once, like letting Joe down the hall read his mail on one node while Jan upstairs is browsing the web on hers. So is the network; it is delivering Joe's mail and carrying HTML traffic to and from Jan's system. These tiny, unpredictable loads make such an architecture less than ideal for fine grained, synchronous calculations where things have to all finish computational steps at the same time or some nodes will have to wait on others.

However, those little loads on average consume *very little* of the computational and networking capacity of the average modern network. There is plenty left over for HPC calculations. Best of all, in many cases this computational resource is *free* in that Joe and Jan *have* to have computers on their desk *anyway* in order to do their work (however little of the total capacity of that computer they on average consume). Most users hardly warm up their CPUs doing routine tasks like text processing and network browsing. They are more likely to consume significant compute resources when they crank up graphical games or

---

get bought to recycle the gold from their contacts, for example. I almost bought Duke's five year old CM5 some years ago for just about this much money but I couldn't figure out how I was going to plug it in in my garage. True story, no kidding, by the way.

[13] *Now* I'm kidding. I just made that up. Cute, huh? Computer folks just *love* jargon and acronyms. Wait for it now...

encode MP3s!

For obvious reasons *many* of the things one needs to know to effectively perform parallel HPC calculations on a true beowulf apply equally well to any OLD network[14]. For that reason, this book is *also* likely to be of use to and provide specific chapters supporting distributed parallel computing in a heterogeneous environment like an OLD network – including those made out of old systems. Even though they don't technically make up a beowulf.

For the rest of the book, I'm going to use the term "beowulf" and "cluster" interchangeably – except when I don't. I'll try to let you know when it matters. If I want to emphasize something that really *only* applies to a beowulf I'll likely use a phrase like "true beowulf" and support this in context. Similarly, it should be very clear that when I talk about a compute cluster made up of workstations spread out over a building with half of them in use by other people at any given time, I'm talking about a NOW or COW but not a "true beowulf".

The beowulf list respects this association – a good fraction of the participants are interested in HPC cluster computing in general and could care less if the cluster in question is technically a beowulf. Others care a great deal but are kind enough to tolerate an eclectic discussion as long as it doesn't get too irrelevant to true beowulfery. Many (perhaps most) of the issues to be confronted are the same, after all, and overall the beowulf list has a remarkably high signal to noise ratio.

Given that we're going to talk about a fairly broad spectrum of arrangements of hardware consonant with the beowulfish "recipe" above, it makes sense to establish, early on, just what kinds of work (HPC or not) that can sensibly be done, in parallel, on any kind of cluster. Suppose you have a bunch of old systems in a closet, or a bunch of money you want to turn into new systems to solve some particular problem, or want to recover cycles in a departmental network: Will the beowulf "recipe" meet your needs? Will any kind of cluster? In other words...

1.2 What's a Beowulf Good For?

The good news is that the "standard" beowulf recipe, as simple as it is, is very likely to result in a beowulf that can accomplish certain kinds of work much faster than a single computer working alone. For that matter, so is any OLD network – for some problems (examples will be given later) the entire Internet can be put to work in parallel on parts of the problem with tremendous increases in the amount of work accomplished per unit time.

The bad news is that the phrase "certain kinds of work" fails to encompass all sorts of common tasks. I really mean it. Only certain *kinds* of work can be run profitably (that is, faster) on a parallel processing supercomputer (of any design).

Even worse, as a general rule a task that *can* be run profitably on a parallel supercomputer will generally *not* run any faster on one unless it is specially

[14]...there!

designed and written to take advantage of the parallel environment. Very little commercial software has yet been written that is designed *a priori* to run in a parallel environment and that which exists is intended for very narrow and specialized applications.

Writing parallel software is not particularly difficult, but neither is it particularly easy. Of course, some people would say that writing *serial* software isn't particularly easy, and parallel is definitely harder. Well, OK, for many people maybe writing parallel software *is* particularly difficult. For one thing, any kind of parallel environment is a lot more complex than the already complex serial environment (which these days has lots of parallel features) and this complexity can interact with your software in odd and unexpected ways.

From this might guess that the design of the *software* to be run on your beowulf-style cluster is likely to be at least as important to the success of your beowulf effort as the design of the beowulf itself. Actually, it is probably *more* important. As this work will explore in great detail, a cost-benefit optimal design for a beowulf can only be determined *after* many of the characteristics of the software to be run on it are known *quantitatively* and an effective parallel software design matched to the *quantitatively* known low-level hardware capabilities.

Although this is a warning, it is not intended to be a discouraging warning. On the contrary, the design above (augmented with a few more choices and possibilities) is remarkably robust. That is, if one has a "big" computing job (the sort that takes a long time to run and hence would benefit from a significant speedup) it is quite likely[15] that it can be rewritten to run (optimally) profitably in parallel on a beowulf built according to the recipe or on a possibly even cheaper and simpler NOW. The remainder of this book is intended to give prospective beowulf builders and users a great deal of the knowledge and design experience needed to permit them to realize this possibility.

With that said, there are most definitely some limitations on this work – there are things it isn't intended to do and won't help you with. At least not yet. Perhaps as the book evolves chapters will be added (by me or other volunteers) to address these topics.

In the meantime, for example, it will say very little about the details of using e.g. PVM[16] or MPI[17] (or any other parallel support library set or raw sockets

[15] The exact probability, of course, is subject to discussion, and therefore has been discussed from time to time on the beowulf list (where we *love* to discuss things that are subject to discussion). Donning my flameproof asbestos suit to repel the "flames" of those that disrespectfully disagree I'd estimate that more than 70% of the computationally intensive work that could be done with a parallel supercomputer of any kind can be done on a relatively simple beowulf design. Actually, I'd say 90% but my suit wouldn't withstand a nuclear blast so I won't.

[16] Parallel Virtual Machine. This is a set of library routines designed to facilitate the construction of parallel software for a "virtual parallel supercomputer" made up of similar or dissimilar machines on a network. It is open source software that significantly precedes the beowulf effort and in some fundamental sense is its lineal ancestor. For more details see the appendix on beowulf software.

[17] Message Passing Interface. Where PVM was from the beginning and open source effort, MPI was originally developed by a consortium of parallel supercomputer vendors in response

themselves) to write a network-parallel program. It presumes that if you are going to write parallel software that you either know how to write it or are prepared to learn how from other resources (some resources will be suggested). Similarly, it isn't intended to be a guide to MOSIX[18] or Condor[19] or any of a number of other parallel computer or cluster management tools.

The one thing it *is* intended to do is to make the reader aware of some of the fundamental issues and tradeoffs involved in beowulf design so that they can do very practical things like write a sane grant proposal for a beowulf or build a beowulf for their corporation (or convince their boss to LET them build a beowulf for their corporation) or build a tiny (but useful) beowulf-style cluster in their homes or offices or out of their existing OLD network.

This latter case is a very important, concrete example of how Linux-based cluster computing can provide near-instant benefits in a corporate or educational environment. Consider the following: One very powerful feature of the Linux operating system is that it multitasks flawlessly as long as the system(s) in question are not constrained by memory. Its scheduler has been deliberately tuned to favor interactive usage[20]. As a consequence, certain beowulf-style cluster designs will allow you to recover the benefit of all those cycles currently being wasted on desktop systems (in between keystrokes while people read their mail or edit some document, or worse, running silly screen savers when the users of the systems aren't even sitting at their desks) *without* impacting the user of the console interface in any perceptible way.

This has to appeal to a small business owner with a big computing job to do and a dozen desktop computers on a network currently twiddling their thumbs

to demands from their clients for a uniform API (Application Programming Interface). Before MPI, all parallel supercomputers had to be programmed more or less by hand and the code was totally non-portable. The life cycle of a parallel supercomputer was: a) Buy the beast. Cost, several million dollars. b) Learn to program the beast. Convert all your code. Cost one or two years of your life. c) Run your code in production for a year or so. d) Realize that a desktop computer that now costs only a few thousand dollars can run your unparallelized code just as fast. e) Sell your parallel supercomputer as junk metal for a few thousand dollars, hoping to break even. I'm not kidding, by the way. Been there, done that.

[18] I have no idea what MOSIX stands for. Perhaps nothing. Perhaps it was developed by a guy named Moe, the way Linux was developed by a guy named Linus. MOSIX software makes your networked cluster transparently into a virtual multiprocessor machine for running *single* threaded code. Very cool.

[19] Don't think Condor means anything either, but the project has a cool logo. Where MOSIX is transparent (attached to the kernel scheduler, if you like) Condor is a big-brother-like batch job distribution engine with a scheduler and policy engine. By the way, anticipating that you're getting bored with pages that are half-footnote, in the future I'll generally refer to software packages without an explanation. As I said before, look in the Beowulf Software Appendix for URL's and more detailed explanatory text.

[20] Linus Torvalds has fairly religiously rejected any redesign of the core kernel scheduler that doesn't preserve "perfect" interactive response within the limitations imposed by the hardware. Philosophically, the Linux kernel is willing to make long-running background tasks wait a bit and perhaps lose 1% of the capacity of the CPU to ensure that it responds to keyboard typing or mouse clicks *now*. As a consequence, a graphical user interface (GUI) user is generally unable to tell whether one, two, or even three simultaneous background jobs are "competing" with their GUI tasks for cycles. You can read the appendix on my own early cluster experiences with a less friendly operating system to see why this is a Good Idea.

(figuratively speaking). It can similarly benefit a university physics or chemistry or math department, where there is a need to cover faculty, staff and graduate student desks with SOME sort of desktop with a web interface and editing/mail tools and there are also significant needs for real computation. If those desktops are Windows-based systems or Macintoshes, their load average is likely to be almost zero – nothing they do on a regular basis requires much CPU – but the unused cycles are wasted as the systems are utterly inaccessible from the network side. If they are running Linux, one can easily run background computations – the whole department can become a readily accessible (transparent to the user) multiprocessor compute cluster managed by e.g. MOSIX or can be running a real parallel (PVM or MPI based) calculation – without the GUI performance of the desktops being noticeably impacted[21].

From this simple example it should be apparent that beowulf-style cluster computing isn't really just for computer scientists or physicists like myself. It can provide real and immediate benefits to just about anyone with a need for *computation* (in the sense of lots of compute cycles doing real calculations) as opposed to an *interface*. Nearly everybody needs to do local computations on local data at least some of the time[22] – beowulfery simply provides an organization with the means to harvest the vast number of wasted cycles that accrue when they *aren't* doing computations locally.

1.3 Historical Perspective and Religious Homage

The concept underlying the beowulf-style compute cluster is not new, and was not invented by any one group at any one time (including the NASA group headed by Sterling and Becker that coined the name "beowulf"). Rather it was an idea that was developed over a long period and that grew along with a set of open source tools capable of supporting it (primarily PVM at first, and later MPI). Note that this is not an attempt to devalue the contributions of Sterling and Becker in any way, it is simply a fact.

However, Thomas Sterling and Don Becker at NASA-Goddard (CESDIS) were, as far as I know, the first group to conceive of making a *dedicated function* supercomputer out of *commodity components* running entirely *open source software* and Don Becker, especially, has devoted a huge fraction of his life to the development of the open source software drivers required to make such a vision reality. Don actually wrote most of the ethernet device drivers in use

[21]This is my own personal favorite approach to parallel supercomputing, and was a major design factor in *Brahma*, Duke's original beowulfish cluster effort which I called a "Distributed Parallel Supercomputer" as it had both dedicated nodes and desktop nodes, mixing characteristics of a beowulf and a NOW. In the physics department we kept Brahma running at over 90% of its capacity for years *and* managed to cover a dozen or so desktops with very nice systems indeed.

[22]Which is why they buy computers instead of "thin" client interfaces, which I personally have repeatedly thought were a dumb idea, every time they've been reinvented over the years. If people (other than corporate bean counters) wanted "thin" we'd still be timesharing with evolved VT100 terminals, for God's sake. It is especially stupid with the marginal cost of "thick" at most a couple hundred dollars. IMHO, anyway.

in Linux today, which are the *sine qua non* of any kind of networked parallel computing[23]. The NASA group also made specific modifications to the Linux kernel to support beowulf design (like channel bonding) that are worthy of mention. Most recently Don Becker and Erik Hendricks and others from the original NASA-Goddard beowulf team have formed Scyld.com[24], which both maintains the beowulf list and beowulf website and has produced a "true beowulf in a box" – the Scyld Beowulf CD – that can be used to transform any pile of PC's into a beowulf in literally minutes.

By providing the sexy name, a useful website, and the related mailing list they formed a nucleation point for all the users of PVM and MPI who were tired of programming in parallel on networks of expensive hardware with proprietary and expensive operating systems (like those offered at the time by IBM, DEC, SGI, Sun Microsystems, and Hewlett-Packard) only to have to buy the whole thing over and over again at very high cost as the hardware evolved. Once Linux had a reasonably reliable network and Intel finally managed to produce a mass-market processor with decent and cost-beneficial numerical performance (the P6), those PVM/MPI users, including myself, rejected those expensive, proprietary systems like radioactive waste and joined with others of a like mind on the beowulf list. This began an open source development/user support cycle that persists and is amazingly effective today.

It is this last contribution, the clear articulation of the idea of the Linux-based beowulf and the focusing of previously disparate energies onto its collaborative development that is likely to be the most important in the long run, as it transcends any particular architectural contributions made in association with the original project. It is an idea that is finally coming to a long awaited maturity – it appears that a number of Linux distributions are going to be providing integrated beowulf/cluster software support "out of the box" in their standard distributions quite soon (really, they largely have for some time, although there have been a few missing pieces). The Scyld beowulf is just the first, and most deeply integrated, of what I expect to become many attempts to make network parallel computing a fully integrated feature of everyday Linux rather than something even remotely exotic.

Beowulfs have always been built from M^2COTS hardware, which is by definition readily available. Soon beowulf support will similarly be in M^2COTS box-set Linux distributions (instead of being scattered hither-and-yon across the web). That takes care of the hardware and software side of things. All that's missing is the knowledge of how to put the two together to make beowulfs work for you, a hole that I'm shamelessly hoping to exploit, errrm, uh, "fill" with this book[25]. With all this to further support parallel development, can commercial-grade parallel software be far behind?

With imitation being the most sincere form of flattery, it is amusing that

[23]Don, et. al. also wrote a book on building a beowulf that is better than mine. So you should buy it. Be aware, though, that it isn't as funny.

[24]See www.scyld.com, duh!

[25]Don't be fooled. I'm in it for the money. Buy this book (even if you're reading a free version from the web). Look, I'll even sign it!

the beowulf concept has been transported by name to other architectures, some of them most definitely *not* open source on COTS hardware (there are FreeBSD beowulfs, NT based "beowulfs", Solaris based "beowulfs", and so forth, where I quite deliberately put the term beowulf in each of these latter cases within quotes to indicate my skepticism that – with the exception of the FreeBSD efforts – the clusters in question could truly qualify as *beowulfs*[26].

Not that I'm totally religious about this – a lot of the clusters I'll discuss below, although COTS and open source, are not *really* beowulfs either although they function about the same way. I *am* fairly religious about the open source part; it is a True Fact that nobody *sane* would consider building a *high performance* beowulf without the full source of all its software components, especially the kernel. I also really, really like Linux. However, even ignoring the historical association of beowulfery and Linux, there are tremendous practical advantages associated with access to the full operating system source even for people with mundane needs.

Issues of control, repair, improvement, cost, or just plain understandability all come down strongly in favor of open source solutions to complex problems of any sort. Not to mention scalability and reliability. This is true in spades for beowulfery, which tends to nonlinearly magnify any small instability in its component platforms into horrible problems when jobs are run over lots of nodes.

If you are foolish enough to buy into the notion that WinNT or Win2K (for example) can be used to build a "beowulf" that will somehow be more stable than or outperform a Linux-based beowulf, you're paying good money[27] for an illusion, as you will realize very painfully the first time your systems misbehave and Microsoft claims that it Isn't Their Fault. They could even be right. It wouldn't matter. It's out of your control and you'll likely never know, since long before you find out your patience will be exhausted and you'll go right out and reinstall Linux on the hardware (for free), do a recompile, and live happily ever after[28]. Use the NT CD's (however much they cost originally) for frisbees with

[26]To quote from the beowulf FAQ assembled by Kragen Sitaker:

> 1. What's a Beowulf? [1999-05-13]
>
> It's a kind of high-performance massively parallel computer built primarily out of commodity hardware components, running a free-software operating system like Linux or FreeBSD, interconnected by a private high-speed network. It consists of a cluster of PCs or workstations dedicated to running high-performance computing tasks. The nodes in the cluster don't sit on people's desks; they are dedicated to running cluster jobs. It is usually connected to the outside world through only a single node.
>
> Some Linux clusters are built for reliability instead of speed. These are not Beowulfs.

See http://www.dnaco.net/~kragen/beowulf-faq.txt

[27]Quite a lot of good money, at that. Goodness, wouldn't you *much* rather give some of that money to me by actually using Linux instead and buying this book with a tiny fraction of what you save?

[28]This is not really a fairy tale. Occurences frighteningly close to this have been reported from time to time on the beowulf list. Remember, for a beowulf to be useful, its nodes have to have a *low* probability of crashing during the time of a calculation, which can easily be days or even weeks. Does this match *your* experience of Win-whatever running anything you

your dog, or as coasters for your coffee cup[29].

At this point in time beowulfs (both "true beowulfs" and beowulf-style M^2COTS clusters of all sorts) are proven technology and can easily be shown to utterly overwhelm any other computing model in cost-benefit for all but a handful of very difficult bleeding edge computational problems. A beowulf-style cluster can often equal or even beat a "big iron" parallel supercomputer in performance while costing a tiny fraction as much to build or run[30]. The following is a guide on how to analyze your own situation and needs to determine how best to design a beowulf or beowulf-style cluster to meet your needs at the lowest possible cost. Enjoy.

like? Enough said. Linux nodes (to my own direct and extensive experience) don't crash. Well, they do, but most often only when the hardware breaks or one does something silly like exhaust virtual memory. Uptimes of months are common – most linux nodes get rebooted for maintenance or upgrade before they crash.

[29] Actually, I favor them for demonstrations of fractal patterns burned into the foil in high voltage discharge involving Tesla coils built by my students. Fascinating.

[30] By "tiny fraction" I mean as little as a few percent. If the "top500" supercomputing list ranked computers in aggregate calculations per second per dollar, instead of just in aggregate calculations per second (hang the cost), big-iron solutions likely wouldn't even make the list. Beowulf-style clusters would *own* it.

Chapter 2

Overview of Beowulf Design

In the Introduction, a simple recipe was given for building a beowulf. In many cases this recipe will work just fine. In others, it will fail miserably and expensively. How can one tell which is which? Better yet, how can one *avoid* making costly mistakes and design an affordable beowulf that *will* work efficiently on your particular set of problems?

By learning, of course, from the mistakes and experience and wisdom of others. This is presumably why you are reading this book. Although beowulf design isn't impossibly complex – beowulfs have been built by high school students, hobbyists, scientists and many others without anything like a degree in systems or network engineering – neither is it terribly simple. It is therefore useful to present a brief overview of beowulf design before we get into the nitty-gritty details that make up much of the next three parts.

Let's begin by setting out a more complex recipe for building a beowulf.

2.1 Beowulf Design Protocol

The primary question to be answered while designing a beowulf is whether or not any beowulf (or cluster, or parallel supercomputer) at all will be useful for solving *your problem*. Some problems parallelize beautifully and simply. Others parallelize well, but not at all simply. *Most* computational "problems", objectively speaking, probably don't parallelize at all, or at least not well enough to benefit from a parallel execution environment that costs any money at all to set up.

For this reason the design process reads a bit like a game – go forward two steps, go back three steps, quit (you lose). This cannot be helped. Engineering is a back and forth process, and even a power hammer like a beowulf simply won't drive a single-threaded screw[1]. The most expensive mistake you can make is building a beowulf for a problem and then discovering that it doesn't speed up the solution at all, or even slows it down.

[1] So to speak...

The following protocol will help you avoid this and other expensive mistakes, and should guide you in your reading of the parts and chapters of this book.

2.1.1 Task Profiling and Analysis

The *first* step in building a beowulf is studying the task you wish to use the beowulf to speed[2]. It really shouldn't be that surprising that the intended function dictates the optimal design, but newbies joining the beowulf list almost invariably get it wrong and begin asking "What hardware should I buy?" which sort of answers itself as the last step of this protocol. The One True Secret to building a successful beowulf, as recited over and over again on the beowulf list by virtually every "expert" on the list[3] is to study your problem and code long and hard *before* shopping for hardware and putting together a plan for your beowulf.

This "secret" is not intended to minimize the importance of understanding the node and network hardware. Indeed, a large fraction of this book is devoted to helping you understand hardware performance issues so you can make sane, informed, cost-beneficial choices. However, it is impossible to estimate how hardware will perform on *your code* without studying *your code*, preferrably by running it on the hardware you are considering for your nodes. In later chapters, concrete examples of code will be given that run at very different relative speeds on a selection of the currently available hardware[4].

The word "study" is used quite deliberately. It means to, if at all possible, use *measurements* and *prototyping* more than back-of-the-envelope estimates[5]. Measurements are far more valuable than any theoretical estimate, however well-informed. A small prototype can save you from all sorts of terrible mistakes, and when a "successful" prototype is finally built, it can can often be scaled up to the final size required[6]

The "study your code" formula above brings to mind a vision of a pocket-protector-loaded geek poring over line after line of program text on green and white lineprinter paper in a dark smoky room with a can of Jolt cola in one hand and a programmer's reference in the other. At least to Old Guys like me. However, this is not at all what I meant. I actually meant one to visualize a pocket-protector-loaded geek poring over line after line of program text in a smoke-free modern linux programming environment with minimally X (and a

[2]We'll refer to "the task" although of course you may well want to build a beowulf to do more than one task. In that case you will need to do most of the work associated with this protocol for *all* the tasks and, if necessary, make trade-off decisions. Beowulves designed for one task won't always do well on another. Be warned

[3]...and a few bozo's, like myself. Oops. Sorry. Inside joke...

[4]This "currently available" hardware, some of it fairly state of the art as of fall 2000, is probably obsolete by the time you read this. Your pocket calculator is likely faster if you're reading this in 2005. Sigh.

[5]You'll have to learn to make back-of-the-envelope estimates, though, so start saving those old envelopes. More on this later.

[6]Noting carefully that scalability of the prototype and your code is one of the things you should be measuring on the prototype, so that the term "often" should properly be "always". Except when it doesn't work, of course.

whole bunch of window panels and desktops), gcc and friends, a debugger or two, emacsoid editors, and/or a ddd-like integrated program environment, with a can of Jolt cola in one hand and the keyboard in the other. Real programmer geeks don't need a hardcopy language reference. That was what should have given it away.

The point is, that to *quantitatively* study your code you have to get *serious* with some of the software development tools that you may well have largely ignored before. I should also point out that even beyond just studying your code, you have to to study your *task*. Even if you have implemented your task in a perfectly straightforward piece of code that a lobotomized lunatic could read and understand, it may be poorly organized to run in a parallel environment. On the other hand, some horribly convoluted rearrangement of the code that you'd never in a million years write in a single-threaded environment (and that a non-lobotomized certified genius might have difficulty understanding) may be just great in a parallel computing environment.

I will now and henceforth assume that you know *nothing* about parallel code design or parallel task execution. Since I (truly) don't know that much more than nothing, I'm going to try to teach you what little I know, and where to learn more. Accept the fact that if you have a "big" project in mind, you will *have to learn more*. I mean it. Real Parallel Algorithms are the purview of Real Computer Scientists (where I am a "Sears" computer scientist at best[7]) and you'll need to find a book by a real computer scientist or two to learn about them. A number of such books are listed in the Bibliography and indicated in the text in context. Alternatively, you can hire a real computer scientist, if can get approval from your fire marshall and the local board of health[8].

Once you have a linux-workstation set up to do the requisite study you can either design a program from scratch to be parallel (a great idea when possible) or, more likely, take an existing serial program and start to parallelize it. To parallelize the program and to inform the beowulf design process, you must begin by identifying how much time is being spent in a serial code description of the task doing work that *could* be done in parallel and how much time is being spend doing work that *must* be done serially. If the linux workstation you are working on is at all "like" what you think you might need for a node (after reading through this whole book) so much the better.

In all likelihood you have no idea how long it takes for your (or any) computer to do *any* of the work in your task. Neither do I. So we must find out. This is accomplished by *profiling* your task. The way to profile a simple serial task using Gnu tools (gcc and gprof) is illustrated in detail in a chapter below.

I'M WRITING RIGHT HERE – THE REST OF THIS CHAPTER IS IN TOTAL FLUX...

Task profiling is covered in a chapter below.

Use Amdahl's Law (covered in a chapter of its own) to determine whether

[7] As well as being an aging Zappa fan...

[8] Just kidding, again, jeeze, can't you take a joke? Seriously, *real* computer scientists are likely to look just like your average, run of the mill pocket protector adorned geek. You can hardly ever pick one out of a crowd of geeks just by looking.

or not there is any point in proceeding further. If not, quit. Your task(s) runs optimally on a single processor, and all you get to choose is which of the various single processors to buy. This book can still help (although it isn't its primary purpose) – check out the chapter on "node hardware" to learn about benchmarking numerical performance.

2.1.2 Parallelizing your Code

2.1.3 Building Price and Performance Tables

2.2 Protocol Summary

1. Profile and Analyze Task(s) for Parallelizability:

   (a) Profile your task(s). Determine how much time is being spent doing work that could be done in parallel and how much time is being spend doing work that must be done serially. Task profiling is covered in a chapter below.

   (b) Use Amdahl's Law (covered in a chapter of its own) to determine whether or not there is any point in proceeding further. If not, quit. Your task(s) runs optimally on a single processor, and all you get to choose is which of the various single processors to buy. This book can still help (although it isn't its primary purpose) – check out the chapter on "node hardware" to learn about benchmarking numerical performance.

2. **Parallelize your code**:

   (a) Just splitting up the serially written routines in your original task (as you probably did to get your first estimate of potential parallel speedup) isn't particularly optimal – one has to adopt "real" parallel algorithms to gain the greatest benefit. Consult one of the resources in the bibliography to learn about parallel algorithms.

   (b) Figure out how much information will have to be communicated between the various subtasks being done in parallel, and estimate how long it take for those communications to be completed with various kinds of networks.

   (c) Parallelizing added some time to the serial and parallel time estimates again, so recalculate them.

   (d) Some of the tasks finish at the same time, some don't. Sometimes we have to wait for everything to be done and resynchronize (at "barriers"), sometimes we don't. Add in and estimate for all the serial time wasted waiting for things to finish up and so forth.

   (e) Use the more detailed forms of Amdahl's Law given in the chapter to determine whether or not there is any point in proceeding further.

If the answer is unequivocally no, then quit. Give it up; use a single processor that is as fast as you can afford or need.

Note that these first two steps can be greatly shortened (to approximately zero) if there already exists a decently written parallel version of your code. Quite a lot of parallel code already has been written by various folks around the world. Look around in the chapter on parallel applications and ask on the beowulf list before tackling a new parallelization project – you might find that your particular wheel has already been invented or that some other wheel can, with a little regrinding, be made to serve as a template for your own.

3. **Build Tables**:

   (a) Build a single node performance table for as many candidate nodes as you wish or can afford, using measurements where you can and calculated estimates where you must.

   (b) Build an estimated multinode performance table for as many networks as you wish or can afford, using measurements where you can and calculated estimates where you must.

   (c) All your answers in the previous step depended on a certain program size. Sometimes making a program "bigger" makes it parallelize more efficiently (explained below in the Amdahl's Law chapter). Determine the approximate effect of scaling up the task size *and* varying the number of nodes in your multinode performance table.

   (d) Attach costs and determine benefits as weights and transform the multinode performance table into a cost-benefit table (covered in a chapter of its own).

4. **Design your Beowulf**:

   (a) Determine your performance threshold.

   (b) Determine your price threshold.

   (c) Select the beowulf (or cluster, or even dedicated parallel supercomputer – if necessary) design that optimizes price performance.

Simple, really. You figure out if your computational task can be sped up "at all" by parallelizing it under ideal circumstances. You figure out pretty much how to parallelize it, up to and including writing or finding or adapting a parallel version of the computational code. You then do a *cost-benefit analysis* of the alternative ways to get the work done[9].

This is a bit oversimplified, of course. For example, the *best* parallel algorithm to choose for certain numerical tasks depends on things like the size of

[9]This isn't just a sane way to design beowulfs, it is a sane way to get nearly *any* major task done. Analyze the problem. Tabulate solutions. Weight solutions with costs, benefits, risks. Choose.

the task and the relative speeds of numerical operations and IPC's, so using one particular one for all the different combinations of node and network hardware isn't likely to give you a global optimum in price performance. The price performance is a function of many variables – the problem itself, the node design (and all the variables that describe a node), network design (and all the variables that describe the network, including e.g. the topology), and the algorithms and methodology used to parallelize the problem. Furthermore, it is a *nonlinear* function of all of those variables – sometimes changing something a tiny bit at very low cost or no cost at all can double overall performance. Or halve it.

Still, the general approach above should suffice to get you into the *vicinity* of an optimum solution, which is generally good enough. In particular, it will give you a very reliable answer to the question of whether or not you can afford to build a beowulf that will run your problem satisfactorily faster. If the answer is yes (and often it will be) then some strategic prototyping and parametric tweaking will often let you do even better.

Chapter 3

Organization of this Book

In order to implement the protocol outlined above, you need to fully understand certain things about parallel programming in general. You also need to know how to quantitatively assess hardware perforamnce, both for the nodes and for the network. The next two parts of the book cover both of these points in detail. The third part of the book addresses the issues of comparative beowulf or cluster design – putting what you've learned about the task, the node hardware and the network together into a plan. This part also discusses potentially expensive design issues that are all too often forgotten – infrastructure costs and requirements, how to systematically compare the cost-benefit of designs, what to do with your beowulf as it ages.

The fourth and fifth parts of the book contain descriptions of and directions to whole "beowulf solutions". Again this is split into chapters on software – discussions of general parallel programming paradigms, examples of specific programs written to do HPC tasks; and hardware – how to craft a beowulf appropriate for different environments and purposes by yourself and how to find somebody who will do it for you (at a reasonable value-added markup).

The appendices and bibliography contain all sorts of useful stuff – URI directions to packages, source for various scriptlets that you might find useful. As this is intended to be a "living book" in the sense that it is continuously updated as new hardware and software appears, appendices may be used as a holding pen for new stuff before it makes it into the "mainstream" text. Even if you own a printed copy of this book, you might periodically check the online master from time to time to see if it has anything radically new.

At this point it is time to begin the serious work. We begin, appropriately enough, with a bit of mathematics. It isn't too difficult, but it is *essential* to the understanding of the principles of parallel program design – and the optimum design of a cluster intended to run the parallel program on.

Part II

Parallel Programs

Chapter 4

Estimating the Speedup: Amdahl's Law

4.1 Amdahl's Law

As noted earlier, the notion of speeding up programs by doing work in parallel (which is the basis of all beowulfery) is hardly new. It dates back to the earliest days of computing, when all computers were so damn slow[1] that people with big pieces of work to do *really* wanted to speed things up. IBM (which made just about all computers at the time and for a considerable time afterward) wanted to speed things up by doing things in parallel (to sell more computers), but rapidly found that there were some fairly stringent restrictions on how much of a speedup one could get for a given parallelized task. These observations were wrapped up in *Amdahl's Law*, where Gene Amdahl was an IBM-er who was working on the problem.

To grok[2] Amdahl's law, we must begin by defining the "speed" of a program. In physics, the average speed is the distance travelled divided by the time it took to travel it. In computers, one does "work" instead of travelling distance, so the speed of the program is sensibly defined to be the work[3] done divided by

[1]They were really *big* and really *slow*. Your pocket calculator or your kid's Nintendo today could probably out-calculate them, and might even have a bigger memory and better programming language. However, Moore's Law, which is another IBM invention, has made computers smaller and faster ever since almost like a law of nature. Moore's Law is discussed a bit later in the text.

[2]Literally, "to drink". See Robert A. Heinlein's "Stranger in a Strange Land". Properly speaking, it would be more correct of you to grok some beer while working on understanding Amdahl's Law. If you have a fridge handy, go on, get a cold one. I'll wait.

[3]Physics purists beware: Yes, this really should be called the "power" of a computer program, not the speed, and power would indeed be a more precise term for what it describes, even though this "work" has nothing to do with force through distance. Work is used in the sense of accomplishing some set of tasks with no proper underlying metric, but ultimately it is related to a sort of free energy.

the time it took to do it:

$$R = \frac{\mathrm{W}}{T} \tag{4.1}$$

where R is the speed (rate), W is the work, and T is the time.

Now we must consider "serial work" and "parallel work". To properly understand all the picky little engineering details of a beowulf we must begin by thoroughly understanding how to "parallelize" a task. Obviously I don't want to talk about parallelizing a computer program as my example. If you already know how to parallelize computer programs, why are you reading this book (except maybe for a good laugh)?

For better or worse, the example task I've chosen to demonstrate parallel and serial work is one that I hope *all* of my readers have undertaken at some point in their lives. It is utterly prosaic, yet it holds important lessons for us. A very Zen thing.

Let us (therefore) imagine that we are *building a model airplane*. Just in *case* some of you have never built model airplanes, allow me to describe the standard procedure. First you open the box. Inside is a set of instructions[4] that are usually numbered 1, 2, 3 and so on. There are three or four plastic "trees" with little-bitty molded plastic parts from which one assembles wheels, wings, propellers or jets, the cockpit, and sometimes a stand[5]. There is usually a sheet of decals as well, that one usually puts on at the end to put strange things like the numbers U94A on the tail and a shark's mouth on the front. One needs to provide a workspace containing a brain (yours)[6] and things like fingers, a sharp instrument for cutting the plastic free from the trees, glue, and possibly paint[7].

Serial computational work is like building the airplane from the pieces, following the instructions one at a time, in order. First we assemble the body. Then we assemble the cockpit. Then we build and attach the left wing. Then we build and attach the right wing. Then we build the tail section. And so on. Sometimes you can do things a bit out of order (build the right wing and then the left wing, for example). Other times building things out of order will leave you pulling the airplane apart to insert the wheel assemblies into the body sockets *before* gluing the body halves together. Occasionally one has to put parts aside and wait for ill-defined periods (like for the glue to dry for some sub-assembly) before proceeding. All of these have analogs in computation. In fact, I'm going to beat this metaphor half to death in the following pages, so I hope you *liked* building model airplanes way back when.

Suppose now that you are building the airplane with a friend. Only one of you can build the body, as there is only one body. Only one of you can build the cockpit. But there are *two wings*...

Aha! We've discovered parallel work. You hand the instructions and parts for one wing to your friend and you both build one wing each at the same time.

[4] *Aha!* he says; the *program*!
[5] The *data*, obviously.
[6] CPU
[7] Tools and peripherals for executing the instructions.

Then you attach them one at a time. If you do *nothing else* in parallel, you complete your airplane in a time shorter than what it would have taken you working alone by (approximately) one wing! Wowser!

Is this the fastest you could have built it with your friend's help? Probably not. There are also several wheel assemblies that could be built at the same time. Also, perhaps you could have worked out of order, with him building the tail or the display rack while you built the cockpit. He might have had to wait a bit for you if he finished earlier, and while he was waiting he might have been able to attach propellers to the engines or attach decals. Or, one might have to put the wings on after the cockpit and before the decals for some reason, so he may have just had to wait until the cockpit was done to do something useful.

If one pauses for a moment to think about it, we automatically parallelize all sorts of tasks in our daily lives as a matter of simple survival[8]. One way or another, a given organization of any task has some time spent performing subtasks that can be done only serially (one subtask after another in just one workspace) and some time spent performing subtasks that *could* be done in parallel (using more than one workspace to independently work on different sets of sequential subtasks). There are obviously lots of ways to split tasks up, and what works best depends on many things.

Running a program on a computer is *also* executing a series of small tasks specified by instructions (code) that cause an agent (the CPU) to act on parts (data) in some environment (the computer) equipped with a set of tools (the peripherals) *just like* building a model airplane. If we identify the serial and parallelizable parts, we can then rewrite our equation for the rate at which a single computer processor can complete a particular piece of work W as:

$$R(1) = \frac{\mathrm{W}}{\mathrm{T_s} + \mathrm{T_p}} \tag{4.2}$$

where T_s (the serial time) is the time spent doing things that have to be done one after another (serially) and T_p (the parallel time) is the time spent doing things that *might* be doable in parallel. The "1" just indicates the number of processors doing the work.

Now, even with "perfect" parallelization and many (P) processors, the program cannot take less than T_s to complete. Even if the entire population of the world were in your kitchen while you were working on the model airplane, you have to complete the body yourself, one step at a time. The best you can do is let all those folks work on the wings and the tail and so forth while you work on the body so as much is done in parallel as possible when the time comes to take the next serial step in assembly. Obviously, the absolute most that having friends help can do for you is reduce the time required to complete the parallel

[8]Think of it as an assignment for the conceptually challenged. I know you're busy, but you're going to have to do your homework if you expect to learn from this course. So put on the stereo, make sure the dinner in the oven, start a load of laundry, pop open a beer, and while all that is going on write down three or four ways you parallelize tasks in your home or office.

work to zero (in fact, you can never quite reach zero). The maximum rate of work one could EVER expect to see with the help of your friends is thus:

$$R(\infty) < \frac{\mathrm{W}}{\mathrm{T_s}} \tag{4.3}$$

Believe it or not, when we add a single variable (the number of friends you have to split up the parallelizable work with), this simple observation is:

Amdahl's Law (Gene Amdahl, 1967)

> If S is the fraction of a calculation that is serial and $1 - S$ the fraction that can be parallelized, then the greatest speedup that can be achieved using P processors is: $\frac{1}{(S+(1-S)/P)}$ which has a limiting value of $1/S$ for an infinite number of processors.

where the result is expressed in *fractions* of the time spent in the distinct serial or parallelizable parts of a calculation:

$$S = \frac{T_s}{T_s + T_p} = \frac{T_s}{T_{\mathrm{tot}}} \tag{4.4}$$

$$(1 - S) = \frac{T_p}{T_s + T_p} = \frac{T_{\mathrm{tot}} - T_s}{T_{\mathrm{tot}}}. \tag{4.5}$$

Since we're going to derive *complicated* variants of this law (that are much better approximators to the speedup bounds) we'd better see how this is derived. We note that the best we can do with parallel work is split it up into P pieces that are all done at the same time so that the ideal time required to complete it turns into T_p/P. The rate of completion of our task with P processors then becomes:

$$R(P) = \frac{\mathrm{W}}{\mathrm{T_s} + (\mathrm{T_p}/\mathrm{P})} \tag{4.6}$$

The "maximum speedup" Amdahl refers to is just the ratio $R(P)/R(1)$:

$$\begin{aligned} R(P)/R(1) &= \frac{1}{(\mathrm{T_s} + (\mathrm{T_p}/\mathrm{P}))/(\mathrm{T_s} + \mathrm{T_p}))} \\ &= \frac{1}{S + (1 - S)/P} \end{aligned} \tag{4.7}$$

making the obvious substitutions for S and $(1 - S)$.

Note again that this is the *best* one can hope to achieve unless the very act of parallelization changes the algorithm used to do the parallel work and/or the hardware interaction into something that reduces T_s (which can happen)[9]. No matter how many processors are employed, if a calculation has a 10% serial component (or $S = 0.1$), the maximum speedup obtainable is 10. If you spend half the time building an airplane doing things that only one person can do, the best speedup you can hope for is for all the other tasks to be done in zero additional time by an infinite number of friends, for a speedup of two.

[9] That is, ahem, (in case you weren't paying attention): "Barring the occurence of a condition such that the law is false, it is true," right? Hmmmm...see the chapter on Bottlenecks.

4.2 Better Estimates for the Speedup

Of course, reality is generally *worse* than the highly optimistic upper bound predicted by Amdahl's Law[10]. After all, imagine how long it would take you to actually build that model airplane if the entire population of the world WERE in your kitchen trying to help. "Forever" might be a reasonable answer as you (and your kitchen) are crushed beneath the weight of all that help. "Forever" is the generally correct answer for parallel processing, too.

There are lots of ways to divvy up the work, and the process of divvying up the work itself takes time. It is also entirely possible to reach fundamental limitations on how far you can subdivide a finite task made of discrete parts (imagine a billion hands on one model airplane that has, after all, only forty or fifty parts that are typically connected by glue bonds made between two objects at a time). There are technical details concerning the order in which subtasks have to be completed that can prevent the parallel work from being cleanly divisible among nodes[11]. And so on. The following analysis works through a few of the simpler and more obvious corrections that we have to consider when parallelizing a task.

One way of looking at all of these corrections is that accounting for all the extra time spent setting up and executing a parallelized task *changes the serial and parallel fractions!* We might expect to see new terms being added to or taken away from the S fraction. Alternatively, thinking about the time it takes to complete the various chores in a team effort, we left out a bunch of *times* that may well be important. Indeed, in many cases (or scales) these additional times may be *dominant* – the *most* important thing that determines the rate or relative speedup.

To understand some of the times we continue with our example of building a model airplane. Note that in our original speedup estimate we ignored the time that it took you to give your friend the wing part of the kit and take back the completed wing. Well, if it takes your friend twenty minutes to build a wing and twenty seconds total to receive the parts and hand back a finished wing, that's probably ok. Your final time estimate is twenty seconds longer than you thought, but compared to twenty minutes that's not too big an error.

What if your friend lives next door and is not in the kitchen with you? You have to get up, take him the wing parts and a tube of glue, come back, and go to work. As soon as he finishes, he has to get up and bring you the finished wing. Now it might take *five minutes* to take him the wing parts and *five minutes* to come back and get to work and suddenly you've spent ten minutes setting up

[10]Except in those very rare cases and small ranges of P where it is better, see previous note. Actually, a bigger danger is that a parallel version of your code will be so different from the original serial version that Amdahl's Law is no longer *relevant* as there is little left of the original "serial fraction" of code. Unless you bother to write a useless "serial" version of the parallel code, you're comparing apples to orangutan's.

[11]Did you ever glue down the cockpit canopy and then learn that *first* you should have glued in the little bitty pilot-in-a-chair assembly?. Oops. The cockpit-canopy gluedown step has to *wait* until the pilot-in-a-chair is done even if it means that you or your friend remains idle.

a parallel process designed to save you twenty minutes. That ten minute net savings might also be relative to an hour's total labor if you did it all yourself. Not too good.

At least you're still in the black – you finish your airplane faster than you would have without your friend. On the other hand, if he lived on the other side of town (a ten minute drive either way) it would take you much *longer* to complete the airplane with his help than it would without it. Whoa! We can actually lose ground and *slow a program down* by parallelizing it! Amdahl's Law (which at least permitted all speedups strictly greater than 1 for any value of P) is *way too optimistic.*

We've discovered a couple of new ideas in our corrected model airplane example, and we'll now proceed to incorporate them into our algebraic discussion of times and rates and speedups and such. The most important is the analogy of "Inter Processor Communications" (IPCs) – this is the communications step where one processor (you) sends part of the program and/or data (the wing parts and glue and instructions for assembly) to another processor (your friend), and later get back a finished wing. In all parallel code, SOME sort of IPC's are necessary. They can take a long time compared to the time actually required to do the parallel work or they can take a short time compared to the time to do the parallel work.

In some very important cases this communication process can be done only one or twice and may take a very short time compared to the time working in parallel, for example at the beginning and the end of a calculation. In all cases, though, the program itself and initial data has to somehow get to the processors working in parallel and the results of their parallel work have to be reunited into some finished whole. IPC's are definitely essential to the notion of parallel processes.

And[12] they take time. And[13] for us to correctly estimate the speedup of a parallelized program, we have to insert the time required, since it may well be significant.

In fact, if we think about it, it costs us time at least TWICE, in fundamentally different ways. Parallelizing a program, we generally increase by a bit the time required to complete the serial fraction. If we have P friends waiting to take various airplane parts that are originally in our possession and build them, we may well have to carry the parts and instructions and glue to each one, one after the other. The more friends, the longer this takes. Overall, this time thus scales like the number of friends (or worse) and, if you are also responsible for doing the serial work it adds directly to the serial time because you're not working on the airplane at all while you're carrying parts around and collecting finished sub-assemblies.

We also have to increase the time required to execute the parallel task on each node a bit over what it would have taken serially. It takes you a certain amount of time (already accounted for in the original serial time estimate) to put

[12] I know, I know, I started a sentence, nay, a whole paragraph, with a conjunction. Again, really. Bad habit. Naughty me.

[13] Oops, did it again. But who cares?

newspapers on the table[14], get the glue open[15], and read the overall instructions – your friend *also* needs to spread his own newspapers on his own table [16], open the glue, and read the general instructions for himself before he can get started on building his wing from the wing-specific instructions. It might take him a few minutes to do these things and we have to increase the time it takes for "wing building in parallel by a friend" over the time it takes for "building each wing as part of sequentially building the whole airplane yourself". This time usually does *not* scale up with the number of friends helping as they can all be spreading newspaper, etc. at the same time. At the same time, it doesn't scale down – your friends can't set up their worksites "in parallel" in less time than it takes to set up a worksite.

Finally, you and your friend will almost certainly have to wait for each other from time to time, as already illustrated above. If you finish one part that gets glued to a part he's working on as the next step, you have to wait for him. Or, you may well be sharing resources. You may have to wait while he uses the glue, for example, and a bit later he may have to wait for you to give it back. In the meantime, you each *may* have to wait idle, although this often depends on how the task is organized.

All this introduces new times and fractions and rates into our earlier statement of Amdahl's Law. Here's a table to help you keep track of this menagerie of variables:

$\mathbf{T_s}$ The original single-processor serial time.

$\mathbf{T_{is}}$ The (average) additional *serial* time spent doing things like IPC's, setup, and so forth, per processor, in all parallelized tasks.

$\mathbf{T_p}$ The original single-processor parallizable time.

$\mathbf{T_{ip}}$ The (average) additional time spent by each processor doing just the setup and work that it does in parallel. This may well include idle time, which is often important enough to be accounted for separately.

The i subscript just reminds us that in many cases the bulk of T_{is} or T_{ip} are due to the burden of either IPC's or idle time, although a lot of T_{ip} can also come from local subtask setup and other non-communications overhead. Note that this is a *simplified* description of the times – in many cases practical discussions of parallel task design split times a bit more finely and specifically and separately count communication, computation, and idle time for all processors.

Using these definitions, we can write our modified task completion time when using P processors:

$$T_{\text{tot}}(P) = T_s + P * T_{is} + T_p/P + T_{ip}. \tag{4.8}$$

[14] If you don't put newspaper on the table first, you're gonna get in trouble when your mother comes home. Remember, I warned you.

[15] I *told* you. It always globs out like that when it first opens. You're gonna be toast.

[16] Presuming, of course, that he has newspapers, a table, and a mother.

In English (for the equation impaired): "The total time required to complete a task that is parallelized on P nodes is the sum of the original serial time, the average additional serial time per node times the number of nodes, the original parallelizable time divided by the number of nodes doing the parallized work, and the average additional time spent on each node to do its piece of the parallelized task". Nothin' to it. We can do this.

To find the Amdahlian[17] speedup, we again have to evaluate $R(P)/R(1)$. This time, being lazy, we'll note that the W always cancels so we're really just evaluating $(T_s + T_p)/T_{\rm tot}(P)$:

$$\frac{R(P)}{R(1)} = \frac{T_s + T_p}{T_s + P * T_{is} + T_p/P + T_{ip}}. \tag{4.9}$$

Now, if we were to be picky (and let's be, just this once) this result, however useful and marvelous, is still way too general (and hence incorrect) to be *truly* useful and marvelous. We are in a bit of a quandry, though. Every time we add a bit of detail, our speedup expression gets a bit more complex. This cannot be helped, it can only be understood, however much work it takes to understand it for your particular numerical task. In *many* cases, this expression will suffice to get at least a general feel for the scaling properties of a task that might be parallelized on a typical beowulf. In others, it won't, and you'll have to work much harder.

As just a single (but very important) example of the latter, it is well-known that certain numerical tasks require a pairwise exchange of information between *all nodes* between parallel steps. Each pairwise communication might take a time T_c (where I have no idea what the c subscript stands for, but it is different from i, s, and p). If there are P nodes and each node can thus talk to $P-1$ *other* nodes and we make the unhappy assumption that they are connected by a *hub* that permits only *one* pair to talk in *one* direction at a time (no broadcasting allowed), we find that the *total* serial IPC step requires $P(P-1)$ individual pairwise communications each costing T_c, or

$$T_{c,\rm tot} = P(P-1)T_c = (P^2 - P)T_c \tag{4.10}$$

which scales *quadratically* in the number of nodes, not linearly!

This, alas, goes in the *denominator* of our relative rate expression, which now contains terms with powers of P that range from -1 to 2. Oooo[18]. Suddenly:

$$\frac{R(P)}{R(1)} = \frac{T_s + T_p}{T_s + P * T_{is} + T_p/P + T_{ip} + (P^2 - P)T_c)} \tag{4.11}$$

and I can just hear the math-challenged among you starting to whimper[19]

[17] Any noun in the English language can be verbed, and in many cases we can invent adjectivish forms as well.

[18] Yes, the term with $P * T_{is}$ is still there. How do you think the program got to the nodes or the results come home at the end?

[19] Relax, the point is that having P^2 in the denominator is baaaaaad if we want to speed things up as P is increased. And those of you who are already parallel computation experts, stop sneering. It's rude.

This is by no means the only kind of scaling behavior possible. If N represents your problem "size" (for example, the length of a lattice side or the number of items in a list to be sorted) then the work being split up can depend strongly on N as well. There may well be (work and/or communication) times that scale like N, other times that scale like N^2, and so forth. If you parallelize the part that scales like N but *not* the part that scales like N^2, you might get decent speedup for smallish N but at some value of N the N^2 will overwhelm it.

Unfortunately, we're just getting warmed up. Imagine what we might have to write if we account for *all* of the times spent in all the parallelized subroutines of a complex piece of code, *including* the effects of nonlinear determinants like cache size, memory speed, memory size, context switching, communications speed, communications latency, communications *pattern*, the need for points of parallel code resynchronization (called "barriers"), and a whole lot more[20]. Each little piece cranks a new term into our relative rate equation, or modifies a term that is already there.

The final insult is that all of these times are totally algorithm dependent and completely different algorithms are often "best" for parallel computations than for serial computations. There are Clever Tricks that can often be used to change the communications pattern and that produce quite different scalings of the communication times and idle times. I *told* you that this sort of thing carries over into the realm of Real Computer Science$_{\rm tm}$ really quickly. Trying to calculate all these terms and times, in detail, *a priori* for complex pieces of code is well-nigh impossible, and few beowulfers (or other kinds of parallel computer programmers) do it. Real Computer Scientists do, it appears (often less for the result than for the papers they can publish describing the result), and bless them for it since then we don't have to.

However, there is no escaping the need to perform a few *basic and practical* steps. The Wise Beowulfer *will* determine the P-scaling and N-scaling of the times of at least the most important blocks of parallelizable code in your task(s) (hopefully, your task will fall into some "generic" category discussed in detail below, but you at least have to identify the category). The Wise Beowulfer *will* at least think about the interaction between your hardware and networking design and algorithm and these powers and times.

Let me conclude this chapter by showing you why you should bother with a couple of practical – if somewhat contrived – examples.

Suppose *you* are charged with building a beowulf to carry out an "all pairs communicate every step" task like the one described above that led to the (somewhat naive) $P(P-1)T_c$ time scaling for "all pairs one at a time" *hub* based communications. Studying the problem, you rapidly learn that you can achieve the same result (all pairs exchanging data) in $(P-1)T_c$ time if a *switch* that can support $P/2$ simultaneous bidirectional communications is used instead of a hub (and P is even). Or you could try using a broadcast on the hub (if your software

[20] Don't worry too much if you don't know exactly what all of these are. I'll discuss them later, at least a little bit.

parallel communications library supports a true hardware broadcast, a thing that might require a prototype to validate since some libraries might implement their group "broadcast" function as a series of pairwise communications to the hosts in the group list), which would yield $P * T_c$ time. You also must consider that T_c might be ten times smaller for a 100 Mbps hub than for a 10 Mbps switch that costs about the same amount. Then there is a 100 Mbps switch to consider, which costs a bit more. Then, we haven't really considered the algorithm. Depending on the message sizes, the latency, and a few very esoteric things, there are algorithms that might reduce the P^2 to (e.g.) $P \log P$ even for a hub (or parallel library) with no broadcast. Finally, we haven't worried about *how* to program a synchronized transfer like the one required to obtain optimum time through a switch. All the nodes have to be talking at the same time and to just the right node, in both directions, to get the improved node scaling, and this is not at all easy to arrange.

Your job, your future, your health and your happiness all ride on making the right choice here. Which one do you buy?

One is tempted to say "the 100 Mbps switch", and for many problems (possibly most) this would be the correct answer. That's why it is in the "recipe" given at the very beginning. It is relatively cheap and adequate to give decent scaling (both power and base time) for many problems. However, if you can only afford eight nodes, and you're doing a problem where $8 * 7 * T_c \ll T_p/8$ for a *10* Mbps hub, the correct answer might be to get the cheapest damn thing you can lay your hands on. There are times when the answer could even be "who needs a hub" – early PC-based "beowulfish" computer efforts not infrequently involved a floppy disk carried around to a bunch of computers *by hand*[21]. Pop in the floppy, merge the data, carry it around, and when it is all shared start another week-long run on all the computers involved.

In quite a few cases, though, the correct answer would be a far more expensive *gigabit* (per second) switch, like Myrinet or perhaps gigabit ethernet. If you want to be able to scale up to "lots" of nodes (say 64 or more) it may be crucial to reduce T_c by an order of magnitude *and* obtain the most favorable P-scaling!

This illustrates just a few of the realities confronting the would-be beowulf designer. In some cases you can derive an approximate scaling form for the relative rate equation appropriate for your particular algorithm and communication pattern. In other cases (most cases, really) you are far better off looking it up (along with a whole lot of other stuff) in a *real* book on parallel computation[22]. The reason you should consider looking things up is because there are smart and stupid ways to do simple little things like multiply matrices in parallel or send a message between all nodes in between program steps. Some of them are very

[21] An early networking protocol known as "sneakernet" that is in not infrequent use even today. My earliest parallel computing efforts used sneakernet.

[22] For example, the one by Amalsi and Gottlieb, from which I cribbed a lot of this stuff. Or there are some great resources on the web, for example the excellent online book on designing parallel programs by Ian Foster at Argonne National Labs, http://www-unix.mcs.anl.gov/dbpp/. This latter resource is particularly awesome (and free) and will be ignored only by the terminally ignorant.

non-intuitive – you'll never invent them on your own[23]. This is what Computer Scientists (the real variety) live for. C'mon, give them their moment of glory.

Once you have the scaling form of the relative speedup appropriate to your algorithm and the various network media types you are considering you can use it (and some measurements) to make *estimates* for the speedup possible for a parallelized chunk of code. This is less difficult than it sounds. In practice, all this mathematical work isn't so daunting – usually most of the parallelizable work is done in just a few program blocks and all the surrounding serial code can be added up at once into the irreducible serial work and the irreducible serial time. In a lot of cases, in fact, there will be just *one* parallelizable block. In the best cases the *whole program* can be converted to a parallel block, where the only required serial code is something to start a lot of programs in parallel and collect the results. These are called "embarrassingly[24] parallel computations".

Although embarrassingly parallel computations are important enough to be given their own acronym (EPC) and to be considered in detail later, we'll take a moment to think about them here as well as they have an important lesson for us to learn before we leave the discussion of mathematical estimates of rates and so forth behind. To understand them we can return to Our Favorite Metaphor by thinking of building *lots* of identical model airplanes with our friends. One can get great parallel efficiency (another way of saying "a speedup like $R(P)/R(1) \approx P$") by just getting P friends into a room[25] and giving each one their own kit and glue. If it takes you ten minutes to distribute 100 kits, and your "nodes" build 100 airplanes in one hour more, you've built 100 airplanes in and hour and ten minutes, for a speedup a hair less than 100. Not bad compared to the 100 hours it would have taken you to build them all one at a time, and you didn't even need to get glue on your own fingers. If you have 1000 friends[26] and can still distribute all the kits in an hour or so (good luck), your gains get even better.

The model airplane construction, in this case, is being run as an embarrassingly parallel task. Now you know what the phrase means. One processor starts P essentially identical jobs (on other processors) all at once, then kicks back for a relatively long time (perhaps sipping a metaphorical brew or two, perhaps doing a job itself) until they all complete, and then collecting the results. Repeat until finished, with near perfect scaling. Technically, we've arranged things so

[23] Unless, of course, you are really very smart or a real computer scientist (in which case you're probably sneering at this miserable excuse for a real book on computer science) or both. I mean, *somebody* invented them, why shouldn't you *re*invent them? Seriously, check out Ian Foster's collection to see why.

[24] One must, of course, get over your embarrassment if your task turns out to be embarrassingly parallel. It's like being embarrassed at being a billionaire or being endowed with a perfect life and great health while others in the world lead flawed lives. A moment or two is all right (to show that you're compassionate), but then it becomes maudlin. Look, the work I do in my physics research is embarrassingly parallel. There, I admitted it. You can too. Let it all out. Maybe we'll start a support group.

[25] Which had better be larger than your kitchen unless P is pretty small.

[26] If you have 1000 friends, of course, you'd never be reading this. You'd either be partying constantly or in politics. Only in the latter case would you be tempted to distribute 1000 model airplane kits, but you wouldn't pay for them.

that T_s, $P * T_{is}$, and T_{ip} are all much much less than T_p/P (with no particularly strong additional constraints on the way tasks are started or finish) so that $R(P)/R(1) \approx P$ as required. This is the way a compute cluster of nearly *any* sort can be used to get fabulous amounts of work done in parallel. Later we'll talk about the SETI project and how to turn the entire internet into a cluster supercomputer.

This embarrassingly parallel example also gives us a hint of how to *improve* our speedup for parallel operation, all things being equal. Suppose we can distribute one airplane kit per minute and need to build ten airplanes. Suppose it also takes only one minute to build an airplane one at a time (perhaps they are the cheap balsa ones your dentist gives to your kids as a "reward" for not biting her fingers). Hmmm, pretty lousy gain[27]. Now, think about the speedup if it takes two minutes to build an airplane[28]. Better, but not spectacular. What about a hundred minutes[29]? Aha! In a lot of cases we can go from pretty shabby parallel speedup scaling to spectacular astounding parallel speedup scaling by just *increasing the amount of parallel work done* while, of course, keeping a lid on the additional serial fraction associated with doing the additional parallel work.

As a parenthetical aside (in a work that a cruel person might consider a huge conglomeration of parenthetical asides [some including nested parenthetical comments of their own] arranged non-parenthetically), one could also be tempted to *reorganize the task completely* from its serial arrangement by setting up an assembly line where each friend just adds one part to a model airplane and hands it to the next person in line. As Henry Ford discovered, such an arrangement requires considerably greater effort (and capital) to set up, but actually can allow the model airplanes to be completed even *faster* than in the embarrassingly coarse grained parallel implementation of the serial work by actively reducing the time required to complete the parallelized work.

Naturally, similar arrangements can occur in parallel programming, especially when considering the additional costs of e.g. flushing and reloading a cache or performing a context switch (which can make it more expensive to do a series of different things in parallel than to do one thing many times). We might even discuss a few later. This is one of several circumstances where Amdahl's Law *might* be wrong, or at least (as previously noted) irrelevant, as there is no useful analog of an assembly line in a non-parallel work situation.

[27] I'm assuming that you are getting out a used envelope, or buying a new one if you have to, and actually working out the relative rates to determine that it actually took 11/10 *longer* than it would have taken you to build them working alone, for a speedup – uh, slowdown – of 10/11. This is known as doing "back of the envelope calculations" and unfortunately you're going to be forced to do this if you hang out with physicists, especially theorists. No self-respecting theorist ever goes anywhere without a pocket full of envelopes. Used is best, but there are few pleasures to compare with filling a crisp new envelope (or the back of your check or the tablecloth itself, in a pinch) with mind-boggling equations and then leaving them on the table for the waitperson to ogle after a four-beer physics lunch. But I digress.

[28] Just in case your envelopes (and brain) are all at home, we get ten airplanes in twelve minutes instead of 20, for a speedup of 5/3.

[29] Sigh. Can't you do the math *yet*? Ten airplanes in 110 minutes instead of 1,000 minutes, or very nearly a speedup of 10 for 10 "processors".

Anyway, I've now completed most of the formal algebraic analysis that I'm going to do. That's the good news. The bad news is that I didn't even try to do a complete or detailed job of the formal analysis – I've only taught you enough[30] that you should be able to *figure out* how to do what you need to do for your own particular task of beowulf design. If your task is complicated enough to be beyond the power of this simple analysis to elucidate (and isn't similar to one of the ones I consider in detail later) then I guess you'll have some work to do, including obtaining and learning from more advanced resources.

There is still one important step to complete before leaving scaling laws completely. Many of you probably looked at equations like (4.9) with the patient, somewhat quizzical expression that I might have if suddenly confronted with a pair of Tibetan monks asking directions to the nearest mall (in Tibetan, of course). I'm so glad you managed to hold on (out of sheer politeness, I'm sure). In the next section we'll actually show the pictures.

4.3 Visualizing the Performance Scaling

If you're the sort of person who is thinking that all the algebraic analysis was just great to read about, but *so* confusing, fear not. There is indeed quite a lot to understand in all of those equations above. For example, examining our basic parallel rate equation (4.9), we see that if $T_{sp} \neq 0$ it will ALWAYS prevent us from profitably reaching $P \to \infty$ for a fixed amount of work.

What, you say? You can't *see* that even if T_p/P goes to zero, the $P * T_{sp}$ diverges, and for some value of P (which we could easily find from calculus, if we hadn't taken a sacred oath not to put any calculus in this discussion) the overall rate begins to degrade?

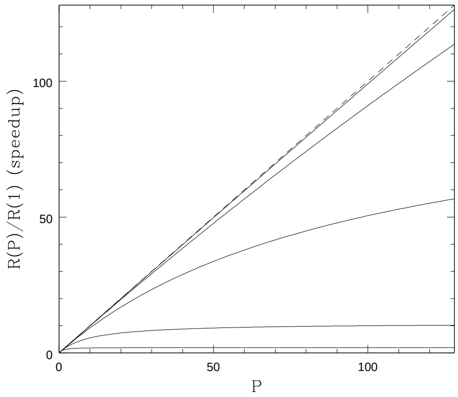
Well, then. Let's Look at it. A figure is often worth a thousand equations. A figure can launch a thousand equations. A figure in time saves nine equations. Grown fat on a diet of equations, gotta watch my figure. We'll therefore examine below a whole lot of figures generated from equation (4.9) for various values of T_s, T_p, and so forth.

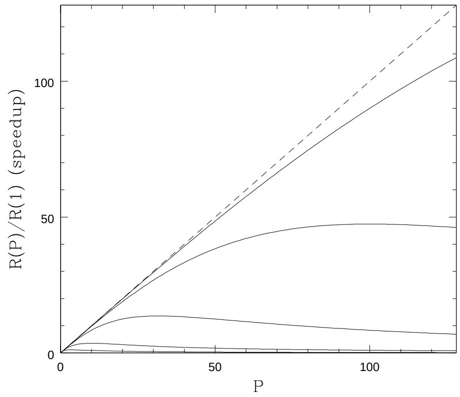
Let's begin to get a feel for real-world speedup by plotting (4.9) for various relative values of the times. I say relative because everything is effectively scaled to fractions when one divides by $T_s + T_p$ as shown above. As we'll see, the critical gain parameter for parallel work is T_p. When T_p is large (compared to everything else), life *could* be good. When T_p is not so large, in many cases we needn't bother building a beowulf at all as it just won't be worth it.

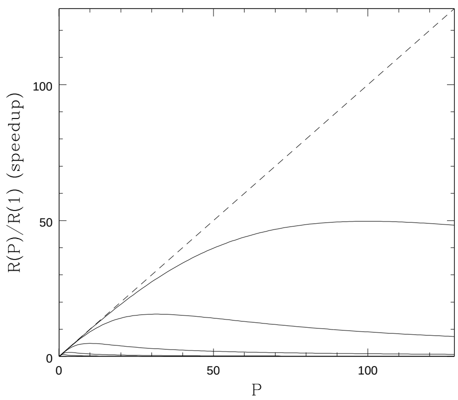
In all the figures below, $T_s = 10$ (which sets our basic scale, if you like) and $T_p = 10, 100, 1000, 10000, 100000$. In the first three figures we just vary $T_{is} = 0, 1, 10$ for $T_{ip} = 1$ (fixed). Note that T_{ip} is rather boring as it just adds to T_s, but it can be important in marginal cases.

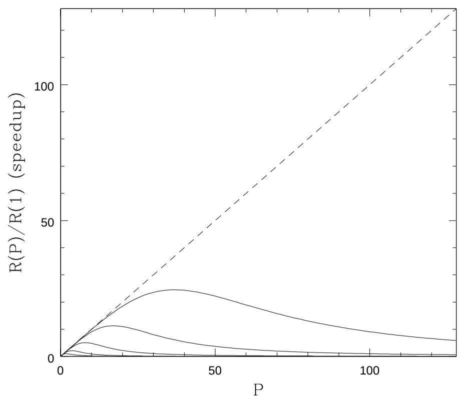
$T_{is} = 0$ (the first figure) is the kind of scaling one sees when communication times are negligible compared to computation. This set of curves (with increas-

[30]Optimistically enough assuming that you've learned all that I've presented so far, of course...


R(P)/R(1) (speedup)
0
50
100
P
0
50
100


R(P)/R(1) (speedup)
0
50
100
P
0
50
100


R(P)/R(1) (speedup)
0
50
100
P
0
50
100


R(P)/R(1) (speedup)
0
50
100
P
0
50
100

ing T_p ascending on the figure) is roughly what one expects from Amdahl's Law, which was derived with no consideration of IPC overhead. Note that the dashed line in all figures is perfectly linear speedup. More processors, more speed. Note also that we never get this over the entire range of P, but we can often come close for small P.

$T_{is} = 1$ is a fairly typical curve for a "real" beowulf. In it one can see the competing effects of cranking up the parallel fraction (T_p relative to T_s) and can also see how even a relatively small serial communications overhead causes the gain curves to peak well short of the saturation predicted by Amdahl's Law in the first figure. Adding processors past this point *costs* one speedup. In many cases this can occur for quite small P. For many problems there is *no point* in trying to get hundreds of processors, as one will never be able to use them.

$T_{is} = 10$ (the third figure) is more of the same. Even with $T_p/T_s = 10000$, the relatively large T_{is} causes the gain to peak well before 128 processors.

Finally, the last figure is $T_{is} = 1$, but this time with a *quadratic* dependence $P^2 * T_{is}$. This might result if the communications required between processors is long range and constrained in some way, as our "all nodes communicate every step" example above showed. There are other ways to get nonlinear dependences of the additional serial time on P, and they can have a profound effect on the per-processor scaling of the speedup.

What to get from these figures? Relatively big T_p is *goooood*. T_{is} is *baaaad*. High powers of P multiplying things like T_{is} or T_c are *baaaaad*. "Real world" speedup curves typically have a peak and there is *no point* in designing and purchasing a beowulf with a hundred nodes if your calculation *reaches* that scaling peak at ten nodes (and actually runs much, much slower on a hundred). Simple stuff, really.

With these figures under your belt, you are now ready to start estimating performance given various design decisions. To summarize what we've learned, to get benefit from *any* beowulf or cluster design we require *a priori* that $T_p \gg T_s$. If this isn't true we are probably wasting our time. If T_p *is* much bigger than T_s (as often can be arranged in a real calculation) we are in luck, but not out of the woods. Next we have to consider T_{is} and/or T_c and the power law(s) satisfied by the additional P-dependent serial overhead for our algorithm, as a function of problem size. If there are problem sizes where these communications times are small compared to T_p/P, we can likely get good success from a beowulf or cluster design in these ranges of P.

The point to emphasize is that these parameters are at least partially under your control. In many cases you can change the ratio of T_p to T_s by making the problem bigger. You can change T_{is} by using a faster communications medium. You can change the power scaling of P in the communications time by changing the topology, using a switch, or sometimes just by rewriting the code to use a "smarter" algorithm. These are some the parameters we're going to juggle in beowulf design.

There are, however, *additional* parameters that we have to learn about. These parameters may or may not have nice, simple scaling laws associated with them. Many of them are extremely nonlinear or even discrete in their effect on your code. Truthfully, a lot of them affect even serial code in much the same way that they affect parallel code, but a parallel environment has the potential to amplify their effect and degrade performance far faster than one expects from the scaling curves above. These are the parameters associated with bottlenecks and barriers.

Chapter 5

Bottlenecks

Or...why Can't Life Be Simple?

Your computer has lots and lots of "moving" parts[1]. They are interlocked in strange and complicated ways. The work that they do for the central processing unit (CPU) proceeds at different rates. Sometimes the CPU has to wait on these processing subsystems. Sometimes the processing subsystems have to wait on the CPU.

You can see that I'm already using the CPU as the lowest common denominator of times and speeds in a system. This is generally the correct thing to do. The CPU clock is generally the fastest one available in the system, and although there is considerable variation in CPU architectures and just how fast they accomplish things, I'm going to assume that on a good day a "typical" CPU executes a single instruction in a single "clock" (cycle) (which is the inverse of the clock frequency). This won't always be true (some instructions may require several clocks, others may complete in parallel in a single clock.

Given CPU speeds that these days range from 300 MHz to 1 GHz, I'm going to assume that typical times required to execute "an instruction" range from 1 to 3 nanoseconds. Linux calls this the "bogus" rate of instruction execution and measures it as a given number of "bogomips" (millions of bogus instructions per second) early in the boot process. It is as good a measure of average processor speed as any other for the purposes of this chapter[2].

Most of the time the CPU (being the fastest gun in the west, so to speak) is the thing that ends up waiting. Whenever we have one part of the computer waiting on another to complete something before it can proceed, we have trouble. If the computer (CPU) is really only trying to do just one thing, it ends up twiddling its metaphorical thumbs until it can go forth and calculate again. If it has other things it can do it can try to improve the shining hour by doing them while it waits. Making all this work efficiently is what multitasking, multiuser operating systems are all about.

[1] Many of which don't actually move, for all that they do a lot of work...

[2] That is to say, it sucks. If you want to know how fast a processor will run your code, run *your code* on it.

When we make an entire network of systems into a computer, this problem is amplified beyond belief. If any *one* CPU is slowed down for any reason (such as waiting on some resource) it can slow down the *whole calculation* distributed on all the nodes. Resources that are likely to be constrained and hence form rate-limiting features of a given calculation are generically called "bottlenecks". It's like having a four lane road that suddenly narrows down to just one lane – the "neck" – traffic often goes even *slower* than it might have if the road had been one lane all along[3].

Let's learn about some of the classic bottlenecks associated with computer calculations – the CPU-memory bus, the CPU's cache (size and speed), the disk and the network itself. We'll also think a bit about how a given parallelized program might have to be written to deal with bottlenecks and (in the next chapter) the related desynchronizing of lots of program elements that can occur if a job is asymmetrically distributed (where one node is faster than another node for whatever reason). The key thing to understand here is that it makes no sense to invest in a figurative ferrari to drive in bottlenecked traffic when kid in a pair of rusty rollerblades might well be able to make faster progress, especially when the money you spent on the ferrari could have been used to widen the road.

To understand bottlenecks on a computer system, we have to first learn the meaning of two words: latency and bandwidth. Latency is the delay between the instant a CPU requests a piece of information and the time the information starts to become available. The word "starts" is key here, as the information requested could be quite lengthy and take a long time to deliver. Bandwidth is the rate at which the information is actually delivered once the delivery has begun. Latency is measured in units of time (typically seconds to nanoseconds) while bandwidth is measured in units of memory size/time (typically megabytes/second).

To give you a really meaningful metaphor, latency is the time it takes from right now to go to the refrigerator and get a beer. Bandwidth is the rate at which (beer in hand) you drink it, as in one beer/hour, ten beers/hour, or so forth. Go on, experiment. Measure the latency and bandwidth of the beer transfer process. Think a bit about the tradeoffs between getting one beer at a time and only chugging it when getting back to your desk (paying the latency of a trip to the fridge over and over) versus getting a whole sixpack in a single trip.

Exciting concept isn't it? Hopefully by now the hangover induced by your experimentation last night has subsided and we can focus once again on our main topic which is beowulfery and not beer (however much they have in common[45]).

[3]This often leads to a certain amount of cursing, some crawl-by shootings, and some good natured road rage. I find that the same is true when one discovers a critical bottleneck in your parallel cluster, especially *after* writing a many-shekel grant proposal to buy it and build it.

[4]If you look carefully, you'll note the word beer is in fact *buried* in beowulfery. Beowulfery!! is an anagram for 'Yow! Beerful!'. Wow. Almost spooky how that turned out.

[5]Or, as Josip Loncaric pointed out, Beowulf is also Foul Web. Don't want to think too

To start with, let's (as is our habit) take a really obvious example so that you see why we're doing this. Suppose that you have a cluster of *diskless* machines available and that each diskless machine has 32 MB of main "core"[6] memory. On this cluster, you wish to run a job that (when running) occupies 64 MB of core (or more) but that partitions nicely into parallel segments that are (say) 20 MB each when running on four nodes.

Running on a single system, the job has to swap constantly, and in this case the only way to swap is over the network to a remote disk on the diskless server. Swapping is *very, very bad* for performance – disks are many powers of ten slower than direct memory access – and swapping over a network compounds the injury. If one has to constantly go to the remote disk to load/unload the contents of memory it can very, very significantly increase the time required to run the program (as I'll discuss below in considerable detail). What is more, it can increase the time required to run the *serial* fraction of your code as much as it increases the time required to run the *parallelizable* fraction, because the parallelizable fraction has to swap in and out where it is interleaved with the serial fraction.

On the other hand, when running on a four node cluster one has to pay the IPC penalty (which we'll assume is fairly small and scales linearly with P) *but* the node jobs no longer swap – they fit into memory with room to spare for the operating system and libraries and buffers and caches and so forth. It is entirely possible that T_s is reduced so much by this that *Amdahl's Law is violated* by the speedup[7].

This shouldn't be *too* surprising. Our derivation of Amdahl's law assumed a certain smoothness of the execution times over the serial/parallel division of a problem; in particular, we assumed that T_s and T_p themselves don't fundamentally change *except* by the division of the parallel work. This not at all unreasonable example shows that this assumption may be *false* in the real world because the system in question has finite resources or resources that are accessible on a variety of timescales.

It also illustrates an important class of jobs for which beowulfs are ideal. One does *not* always consider building a beowulf to achieve a "speedup" in the ordinary sense of the word. One can also build one to enable a job to be done *at all*. More realistically (since in the previous example an obvious solution is to invest \$30 in another 32 MB of memory) if one has a job that runs in ~ 5

hard about that one...

[6]You can always tell an Old Guy in computing because we still use quaint terms like "core" to describe something that hasn't been an actual core since before most of the folks reading this were born. I've actually seen and held in my hands antique memory cores, which looked like a funny 3-dimensional grid of wires and beads in a tube. Chip-based memory is boring by comparison. You can see a picture of a memory core (complete with a nifty "magnifier" that scans the beads) at http://www.physics.gla.ac.uk/~fdoherty/IDRG/lense.html at the time of this writing.

[7]If one is inclined to argue, imagine the speedup if the four nodes in question *don't* have remote swap or any swap at all. In that case the job takes an infinite amount of time on a single node as it just won't run, and will take a finite amount of time on four nodes. How's an *infinite* speedup for a violation, eh?

GB of memory it may be far cheaper to purchase ten 512 MB systems than one 5 GB system, presuming that one can find a system at any price that holds 5 GB of main memory[8]

Speedups that violate the simple notions that went into Amdahl's Law or the slightly more realistic speedup equation (4.9) are called *superlinear speedups*, and a vast literature has developed on the subject. Basically, a superlinear speedup is what we call it when for any reason parallelizing a program results in a speedup that scales faster than P for any part of its range. In nearly all cases, these will occur because of the wide range of timescales available within "a system" for *accessing data or code*.

Bottlenecks (in this case the bottleneck associated with disk-based virtual memory) can clearly wreak havoc on our beautifully derived speedup expressions, for good or for ill. Let's take a quick look at the primary bottlenecks that can significantly impact our parallel performance. I'll try to include some simple code fragments that either illustrate the points or permit you to estimate their impact on your code.

We'll begin with a simple table of the bottlenecks and the sorts of (both practical and theoretical) limits associated with them:

Bottleneck	Description	Latency	Bandwidth
L1 Cache	CPU to L1 Cache read/write	1-5 ns (1 clock cycle)	-
L2 Cache	L1 to L2 Cache read/write	4-10 ns	400-1000 MB/sec
Memory	L2 Cache to memory read/write	40-80 ns	100-400 MB/sec
Disk (local)	CPU to disk read/write	5-15 ms	1-80 MB/sec
Disk (NFS)	CPU to NFS disk read/write	5-20 ms	0.5-70 MB/sec
Network	CPU to remote CPU write	5-50 μs	0.5-100 MB/sec

The first thing to note in this table is the times therein differ by seven or more orders of magnitude for different devices. Compare 1 clock tick (as little as 1 nanosecond for a 1 GHz CPU) latency accessing a particular address in the L1 cache to 10 or more *milli*second latency accessing a particular address on a hard disk. Compare bandwidths of a megabyte per second for streaming data transfer over a slow network (often degraded to half that or even less) to rates on the order of a gigabyte per second from the L1 cache. Even this fails to encompass the full range – we don't even consider floppy drives, serial networks, or tape devices on the slow end or the internal rates of register activity within

[8]As I discuss later below, running the job on a 512 MB system with 5 GB of *virtual* memory in the form of swap could conceivably transform a job that would take a mere day or two on a system with 5 GB of memory into one that would probably finish just in time for the next millenial celebration. No kidding. Six orders of magnitude will do that to you.

the CPU itself on the high end.

The second thing to note is the the entire reason for having a full hierarchy of "memory" access rates is to *hide* the longest times and slowest access rates from you. Your *average* data access rate and latency is determined by how often the data you need is available in each different kind of memory (where we will consider disk to be a kind of memory, and will often think about network reads or writes in terms of memory as well). Time for an equation. Suppose p_i is the probability that your program finds the next piece of data or code it is looking for in the ith kind of "memory" in the table above. Then the average latency (the average time the system has to wait before the data or code becomes available) is:

$$< L >= \sum_i p_i L_i. \tag{5.1}$$

For example, if $p_{L1} = 0.05$, $p_{L2} = 0.9$, and $p_M = 0.05$ for a problem that fits into main memory, the average latency might be something like:

$$< L >= 0.05 * 1 + 0.9 * 8 + 0.05 * 50 = 9.75 \tag{5.2}$$

in nanoseconds. Things get a little more complicated trying to determine the overall data access rate (I'd rather not get into a full discussion of EDO, SDRAM, Rambus, and so forth at this moment and instead will just give you a web reference[9]. However, the bottom line is that as long as the hierarchy of your hardware accomplishes this efficiently (maintaining an average latency and bandwidth reasonably close to that of the L1 and/or L2 cache) *for your code* there is no reason to invest in a more expensive hierarchy.

The following is my own, strictly editorial opinion[10]. To put it bluntly, the advantages of a large L2 cache are often overblown by CPU manufacturers interested in selling you larger, more expensive (and profitable!) CPUs. It is amusing to compare the actual execution times for given pieces of code on similar clock processors from the Intel Celeron/PII/Xeon/PIII family. The Celeron, Xeon and PIII have caches that run at the full speed of the CPU, so that on an (e.g.) 500 MHz CPU a clock tick is 2 nanoseconds. The PII has a cache

[9]See http://pclt.cis.yale.edu/pclt/PCHW/CPUMEM.HTM

[10]One which it is always nice to see is shared by others. For example, a quote from the aforementioned web page on memory hierarchy:

> All other things equal, more cache is better than less. Clearly no desktop user is going to blow $3800 to get a Xeon processor that, for ordinary applications, will be almost indistinguishable from a $150 Celeron. Just because a Pentium III system is within reach, is it really worth the money?
>
> Vendors have an incentive to sell more expensive units. Some customers will opt for the more expensive machine because they think they're worth it. However, most casual users will get along quite nicely with a Celeron. The Pentium III is engineered best for a workstation or server with two CPUs. If it ever makes sense, the Xeon is designed for corporate servers with 4 or 8 processors.

This is part of the Yale "PC Lube and Tune" website, which is actually rather nice and well worth bookmarking as an extended resource: http://pclt.cis.yale.edu/pclt/default.htm that runs at half the speed of the CPU clock. The Celeron L2 cache is 128 KB in size; the PII and PIII caches are 512 KB in size (four times larger) and the Xeon cache comes in several sizes, but can be as much as 2 MB in size if you're willing to pay an absurd amount of money for it.

For "most code" (where once again I risk flames with such a fuzzy term) there is little benefit to be seen in having a CPU with the larger cache sizes. Given a factor of ten or so cost differential between the small-cache Celeron and a very large cache Xeon at equivalent clock, one can afford to buy *three complete Celeron systems* for what it costs to buy *one 2MB cache Xeon processor* and only rarely does one see as much as a 20% speed advantage associated with the larger cache. However, there are exceptions. Code/data sets that fit within a cache clearly will execute far faster than code/data that has to go to main memory (see table above).

The reason that caches tend to work so well is that in a *lot* of cases a relatively few instructions are executed sequentially in loops, so that loading a whole block into cache from memory just one time suffices to allow the program to run out of cache for extended periods. If one is very lucky, one's core code can live in the L1 cache and runs at "full speed" all the time. Similarly, a lot of time the data one works on tends to be "localized" in the memory space of the program so that one load works for an extended period. If you like this makes $p_{L1} + p_{L2} \gg p_M$ so that, on average, the CPU finds what it needs already in the cache with occasional long delays when it doesn't and has to reload. As we've seen, this tends to yield and average latency and access speed not too far from the bare speed of the L2 cache which keeps your CPU trucking right along doing useful things instead of waiting for data.

However, if your program does things like add up randomly selected bytes of data from a one megabyte dataspace, it may *not* find the next byte that it needs in cache. With a 128 KB L2 cache, the probability may be no greater than 1/10 that it does, so 90% of the time it will take some 60 nanoseconds to get the next byte to add, and 10% of the time it will only take 10 nanoseconds (or less) for an average rate of one byte in 55 nanoseconds (plus a tick or so for the add). With a 512 KB L2 cache, it might find the byte it needs (after the program has been running a while) 1/2 the time, and its average rate of access goes up to one byte in 35 nanoseconds. With a 1 GB L2 cache, the data lives entirely without the cache and the program can get the next byte in 10 nanoseconds, nearly six times faster than with a 128 KB cache.

Some tasks are "like" this and have a very nonlocal pattern of memory access. These tasks benefit from a large cache. As you can see from the numbers above, though, if one finds 90% or more of what one needs in L1 or L2 cache already, there is little marginal return paying quite a lot of money for having a bigger one.

Note that this kind of problem can easily exhibit a nice superlinear speedup on a beowulf. If one breaks one's 1 MB into ten pieces 100 KB each in size, they will run in cache on Celeron nodes for a speed advantage (per node) of nearly six *in addition to* the factor of 10 for parallelizing the sums. If the blocks can be independently sampled and summed for a long time (to minimize the relative IPC cost of transferring the blocks and collecting the partial sums at the end) one may see a speedup far greater than one might expect from just using 10 nodes to do something you were before doing on one and ignoring the cache interaction. A silly example, in that I can think of no useful purpose to summing random bytes from a given memory space, but there are likely useful things (perhaps in a simulation of some sort) with a similar access pattern.

Now, with my editorial comment completed, I do not mean at all to suggest that the speeds and latencies of the memory subsystems are unimportant to the beowulf designer – quite the contrary. There are many jobs that people run on computers (beowulf or not) that are "memory bound" (which just means that their speed is primarily determined by the speed with which things are retrieved from memory, not the speed of the CPU per se). Multiplying very large matrices and other sequential operations involving large blocks of memory are perfect examples[11]. In many of these operations the system is "always" getting new blocks of data from memory and putting them into cache (or vice versa) so that the memory subsystem is more or less continuously busy.

In that case an important new bottleneck surfaces that is a frequent topic of discussion on the beowulf list. It is well known that the cheapest way to get CPU is in a dual packaging. Dual CPU motherboards tend to be only a tiny bit cheaper than single CPU motherboards, and a dual can share all other resources (case, memory, disk, network) so you only have to buy one set for two CPUs. In one direction, the marginal cost of a dual over a single is the cost of a second CPU plus perhaps $50-100 (in the case of Intel processors – YMMV). In the other direction, the marginal cost of a second single CPU node compared to a dual node is the cost of a case, memory, disk and network (less $50-100) or perhaps $100-300, depending on how much memory and disk you get.

If your calculation is "CPU bound" then a dual is optimal and your beowulf design should likely be a pile of duals. In many cases EPC's will be CPU bound – more CPUs means more work done. If it is memory bound, however, it is a true fact on Intel systems that duals more than saturate the memory subsystem. If two CPUs are trying to get things from memory at the same time as fast as they can, one CPU has to wait at least a fraction of its time. This can impact memory bound performance significantly so that instead of getting 200% the performance of a single CPU system, one gets only 140-160%. In this case, one is usually better off getting two singles (which can yield the full 200%).

If your calculation is network bound (a possibility discussed in detail in the next chapter) life becomes far more complicated. In that case, there are lots of possibilities to consider including communication pattern, putting two NIC's in one case, being effectively memory bound (one generally talks to the

[11] It is worth mentioning that linear operations like this can be very significantly sped up (by a factor of 2 to 3 or more) by precisely organizing them to match the actual sizes of the caching subsystems. ATLAS (Automatically Tuned Linear Algebra System) is a project that has written some very clever code-building code that creates linear algebra libraries (BLAS and LAPACK) with loop and block sizes precisely tuned to cache size to yield empirically the best possible execution times. Very, very cool. See http://www.netlib.org/atlas/index.html to get the package.

NICs through the memory subsystem) and the fact that a dual can in some circumstances use a network more efficiently than a single because *receiving* a network transmission turns out to proceed much, much more slowly if the receiver's CPU is busy running code. I therefore hesitate to give a general rule for singles versus duals in situations where your code is network bound – you're better off prototyping your code and recycling the losing hardware on desktops or as servers.

All of the arguments and discussion concerning the L1-L2-main memory bottlenecks hold true when extended to jobs that swap, only everything becomes much, much worse. When a job swaps, an even slower "memory" (the hard disk) is used to store part of the (virtual) memory image of the running job. Under the assumption that the code and data are reasonably local, pieces of them are loaded into real memory on demand (whenever a virtual address is requested that is on the disk instead of in real memory) usually in fairly big chunks (pages). In fact, the system does this even when it doesn't "have" to and typically keeps only what it is actually working with in memory to conserve memory for all sorts of buffering and caching optimizations that the operating system handles for you behind the scenes.

The reason for big chunks is that if you have to pay that hideous 5-10 millisecond latency (often twice!) to get any chunk at all, you have to get a "big" one to keep the average transfer rate from being absurdly low. You're also betting (literally) that your next data or code requirement is more likely to come from the big block you just read in than not.

You pay the penalty twice when you have to store some of the pages in memory to disk to liberate the space for new pages coming in from disk. This generally happens for data, hence the term "swap", where two data pages are exchanged. Code, on the other hand, tends to be read in single pages from the single fixed disk image of the binary and its associated libraries, where the system works hard to cache frequently accessed pages to avoid having to actually use the disk. The two kinds of virtual memory operation (page and swap) are accounted separately in /proc/stat – lots of paging is normal, lots of swapping (or even any swapping at all) is dark and evil. Even reading in large chunks, the many orders of magnitude difference in writing to and reading from the disk instead of memory is very, very costly to a program's speed.

As (5.1) shows, if one is lucky and the code and data references are indeed mostly clustered, a job can swap or page and still complete in a reasonable amount of time. The calculation looks very similar to the example above except now one has an additional term where one has to multiply the probability of having to go to swap times the rate (including the combined effects of latency, transfer bandwidth, and the size of the block requested) to the other terms. As long as the vast bulk (as in more than 99%) of requests are satisfied from CPU cache or main memory, an occasional swap or page isn't too painful.

Life starts to really suck, however, when this is not true. If we extend the random access example to swap-based virtual memory, we can arrange things to deliberately defeat the best efforts of the paging and swap algorithms and force the system to disk again and again. For example, on a system with only N bytes of RAM, one can create a job that occupies a $10*N$ virtual memory space, $9*N$ of which necessarily reside on disk. Sequentially adding randomly selected bytes of data from this long vector will force the system to memory (instead of cache) on almost every call and on to disk 90% of the time, paying approximately (0.9*10 = 9) milliseconds per add. Adding a mere 10^9 numbers would require some 10^6 seconds, or almost two weeks. Compare that to adding 10^9 numbers selected from a vector that fits in L1 (on the order of a few seconds) or L2 (a bit less than a minute) or memory (a couple of minutes). You can see why understanding the bottlenecks associated with the different speeds of the various memory subsystems is so important when engineering a standalone workstation, let alone a beowulf cluster.

Again you can see how a 10 node beowulf design that permits the task to execute out of main memory could yield a millionfold improvement in time to completion, which is a rather profound nonlinear speedup. This also justifies the earlier observation that a job that might run in some reasonable fraction of a day on such a beowulf could be stretched to 365000 days and end in early 3000 A.D. on a single node with swap. Disk is very, very slow compared to any sort of "real" memory subsystem and both tasks and the beowulfs they run on should always be designed and tuned to avoid requiring swap at all costs.

However silly this example appears at first glance, there are a number of tasks with a similar lack of locality that do, in fact, occupy very large virtual memory spaces. Sorting very long lists, database operations, certain kinds of simulations, all might perform operations on very widely separated or even randomly selected elements in a list. Here is another place where algorithm becomes almost as important as architecture – some algorithms for a sort, for example, might be far more local than others and although they may scale worse in terms of number of operations, by avoiding a killer bottleneck (tending to run in cache, for example) they may complete in far less time.

It is interesting to note in the context of beowulfery that (following the table above) it is some two to three orders of magnitude faster to transfer data from the memory of a remote system over the *network* than it is to transfer it from a *disk* (whether local or remote). Even if one's job isn't capable of being partitioned among nodes, if it requires four GB of virtual memory (and all you have is 512 MB nodes) one can obtain nearly a thousandfold speedup compared to running out of swap by putting swap spaces on remote ramdisks on the nodes that are then served to the single-threaded task execution unit over the network. In principle this can be done with current linux kernels (make a big ramdisk on a node, build a swap space on it that is provided via NFS to the execution node) but I haven't tried it. It is likely that (if it works at all) it isn't really very efficient, however much it improves on a disk based swap.

This is an area of current research by real computer scientists. The Trapeze project being conducted (in some cases by friends of mine) at Duke[12] is one such effort (based, alas, on FreeBSD) that uses e.g. Myrinet as the network layer. With its ~ 5 microsecond latency and gigabit per second bandwidth, Myrinet is

[12] See http://www.cs.duke.edu/ari/trapeze

fast enough to form the basis to an intermediate layer in the memory hierarchy directly integrated with the kernel, rather than operating through the usual swap or paging mechanism. Again, the point is that beowulfish architectures can provide tremendous nonlinear speedups and enable new work to be accomplished at far lower cost than (for example) buying a system equipped with four or five gigabytes of main memory.

The network is such an important bottleneck in a traditional "real" beowulf calculation that it deserves a section all its own. In the next chapter we'll examine the network and IPC's in a beowulfish layout. We'll also show how IPC's and the not infrequent requirement that your code proceed by parallel steps synchronously can combine to push your optimal design towards, or away, from a "true beowulf" configuration as opposed to a generic cluster.

There are several bottlenecks that I haven't discussed in this section that may or may not be important to your code. For example, I've only barely mentioned context switches[13] without telling you what they are or why they are "bad". They are what happens when your code bounces around in certain ways and forces the cache to be reloaded with new code, and they can occur when you call lots of widely separated subroutines or when you access the I/O subsystems a lot, among other places. So try to avoid doing this sort of thing inside your main loops, all right? I also haven't talked much about interrupts per se, partly because interrupts and context switches live in the Deep Kernel and are Not Meant for Mere Mortals to Know. Or something like that.

Actually, interrupts are pretty important to beowulfery, especially those associated with the network, again because there are all sorts of nasty latencies and bandwidths and contention issues to deal with in exotic circumstances. However, most of this will be largely beyond one's control unless one happens to be a kernel hacking kind of person with an I.Q. of around 170[14]. So a really proper treatment of this will have to wait until I write a book on the kernel, which, given my truly astounding lack of detailed knowledge of how the kernel works, could be forever[15].

[13] Right here, in fact.

[14] Or happens to be a somewhat stupider pocket protector clad geek type who never sleeps or bathes because one spends one's life studying the kernel.

[15] Proving, for the logicians amongst us, that I either am not a kernel-hacking kind of person, my I.Q. is less than 170, or that I at least occasionally bathe and sleep.

Chapter 6

IPC's, Granularity and Barriers

OK, by now you should be getting the hang of things. A beowulf is a parallel supercomputer built out of COTS nodes interconnected by a COTS network of some kind. One can build a beowulf to speed up a piece of parallelized code (in the classic Amdahlian sense) so it finishes faster. One can build a beowulf to be able to do a task at all by assembling more resources than one can either afford any other way or than are currently available in a system at any price. One can build a beowulf to speed up a code in an exotic way (by providing a faster extended virtual memory space, for example).

In the previous chapter we discussed all sorts of ways the basic bottlenecks between the CPU and memory subsystems (within a node, by assumption) can affect program speed, trying to provide a semi-quantitative understanding so that you can at least do the back of the envelope calculations required to compare the cost-benefit of various alternative ways of accomplishing a task. In this chapter we'll focus on the *sine qua non* bottleneck of beowulfery, the network.

There is so *much* to learn about networking and how it relates to serious beowulfery that it is hard to know just how much to put into an introductory book like this. To invert the point, there is such a wide range of ignorance about networking out there that I could easily be speaking to someone who doesn't know Appletalk from Ethernet, has never heard of the ISO or OSI, for whom TCP and IP are a mystery, and who thinks that a router is a device for cutting interesting curves in a piece of wood.

If this is you: Sorry, chum, you won't learn about these things here, or at least you won't learn much (certainly not enough to assemble a functional linux network). What can I say – there are whole books that focus on *just* setting up and running a network, and I cannot compress all that into a chapter and have time to say anything at all about networking in the fairly strict context of beowulfery.

So, even though a network is key to a beowulf, I'm going to assume that in fact you do know what the following are:

- NIC (Network Interface Card), typically a PCI (regular, fast, or gigabit) ethernet adaptor or PCI Myrinet adaptor, although there are now some exotic alternatives with more on the horizon. This little pup plugs into the PCI bus of your node and is connected via RJ45 cables to an appropriate
- Hub. This is something that distributes a signal in the transmit wires of one connection to the receive wires of all the other connections. There are all sorts of technical things associated with hubs (like the number of hubs you can put in between hosts). In general, hubs are "bad" in beowulf design unless one's needs are modest and likely to stay that way. If one can afford it, one would usually do better with a
- Switch. This is something that establishes a virtual pairwise connection between hosts plugged into the switch. Where the hub allows only *one* host at a time to talk in *one* direction, a switch allows *all* the hosts to talk at the same time in *both* directions. In principle – be sure to check the "bisection bandwidth" of your switch to ensure that it can indeed handle full-duplex on all lines at once as a cheap switch might not.
- Ethernet. The most common (and cheapest) networking hardware protocol. Ethernet comes with an amazing and complex set of rules and standards, such as how long a cable can be used to connect a host to a hub or switch, how many hubs can exist between hosts, what an adaptor has to do in the event of a collision (what a collision is in the first place), in addition to certain baseline latencies and bandwidths and packet sizes. Can't know too much about ethernet, no sirree-bob!
- Myrinet. The most common and probably premier gigabit network. It's about ten times faster than 100BT ethernet, and costs about twenty times as much per node (or even a bit more).
- IP. "Internet Protocol". Yes, it is the fundamental protocol upon which the internet is founded. Associated with IP is the "internet address" (or IP address) of a host (adaptor), a packet structure (headers and so forth), and above all a routing and filtering mechanism.
- TCP. "Transmission Control Protocol". This lives "on top of" IP and regulates things like the reliable delivery of packets across an uncertain network. Most common services (like mail, ftp, telnet, http and so forth) are defined in terms of standard port addresses within TCP. TCP is a bit smarter and more reliable than "raw" IP, and hence is a tiny bit slower.
- Sockets. A "socket" is an abstraction for a network connection. One opens up a socket on a host and a remote service connects to that socket. Information flows between the sockets. Sockets can be read or written to like files (of course, in Unix and hence linux, everything is a file, right?).

- UDP. "Unix Datagram Protocol". This is a kind of socket and hence network connection. It is the "raw IP" connection I referred to earlier. A TCP socket is more reliable but has certain limitations and costs. UDP is commonly used to provide local services within a local area network LAN), TCP across a wide area network or WAN. Except that isn't necessarily true anymore, as reliability is almost always more important than raw speed. NFS is probably the most famous UDP service.

- Router. Something that routes *packets*, usually IP packets (actually a router can often route lots of kinds of packets but we don't care about any other kinds).

- Gateway. A router that typically lives between an "inner" network (LAN) and an "outer" network (ultimately the rest of the Internet). It lets in the good packets and lets out the bad packets. A true beowulf typically has a gateway node that is also usually a "head" node from which it is controlled and may also be a server node if the internal nodes require (e.g.) NFS services. The gateway node can keep spurious external traffic off the private internal network of the nodes.

As you can see, I am omitting all sorts of useful and important things. You won't learn about netmasks, broadcasts, how to configure a NIC, or any of that from me. However, I will direct you to the /usr/doc/HOWTO directory (in most linux distributions) that has explicit step by step instructions for setting up all sorts of things including the network. Don't forget about Linux Headquarters (http://www.linuxhq.com/) either, which has links to all the HOWTOs and other documentation. There are a bunch of key learning documents in my own personal website including http://www.phy.duke.edu/∼rgb/security/local.guide, which is "the" classic 1988 Rutgers white paper by Charles Hedrick describing all sorts of networking concepts. Finally, there are a whole bunch of useful URL's on the Brahma website (http://www.phy.duke.edu/brahma) which might be of interest to the neophyte.

SO, from here on I'm going to assume that you can design and set up a simple ethernet-based IP subnet without having your hand held. We'll still address some of this sort of thing in the next chapter, but for now we'll focus on the technical details (especially things like latency, effect of packet size on bandwidth, problems, solutions) and not on truly introductory things.

6.1 Shared Networks

6.2 Switched Networks

Chapter 7

Profiling

- To see if a beowulf makes sense, we therefore must *begin* by determining T_s and T_p (for a given amount of work W and total execution time T).
- For example, word processors are almost entirely serial; $S = T_s/T \approx 1$ and are I/O bound as well (we'll get to this later). It would be stupid to build a parallel word processor.
- Many statistical simulations, on the other hand, can be run "completely" in parallel, with a relatively tiny fraction of the code serialized to collect results. For these $S \ll 1$ and it is easy and profitable to run in parallel.
- Compile with (gcc) -pg compiler flag, use gprof to see where program does the most work, identify parallelizable sub-tasks. Or use BERT (Fortran), or other similar tools. It T_p (even under ideal circumstances) worth it?
- Note: "Larger" problems can be attacked by a beowulf than on a UP system, which may make them worthwhile even when the scaling is lousy.

Parallelizing the Discover$_{\rm tm}$ Neural Network
An Example

Run gprof on discover building a simple neural network (ten bit "divisible by seven"). Small training set, not many neurons. We get:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 38.53     20.84    20.84 67549440     0.00     0.00  act_func
 34.81     39.67    18.83 67549440     0.00     0.00  dotprod
  8.36     44.19     4.52 67549440     0.00     0.00  activity
  6.84     47.89     3.70 13305088     0.00     0.00  trial
  4.66     50.41     2.52     4052     0.62     1.65  find_grad
  3.29     52.19     1.78    47919     0.04     0.95  eval_error
  1.72     53.12     0.93      800     1.16     1.19  dsvdcmp
  0.89     53.60     0.48  5186560     0.00     0.00  actderiv
  0.30     53.76     0.16      800     0.20     2.27  regress
  ...
```

trial, act_func, activity, and dotprod are all used to evaluate the *training set error* of a neural network. Together they comprise more than 80% of the code. If we can parallelize the evaluation of training set error, we can expect a fivefold or better speedup for the run I profiled.

Or can we....?

The answer, of course, is NO – this is just an upper bound and one that depends strongly on problem size at that. Still a useful case to investigate.

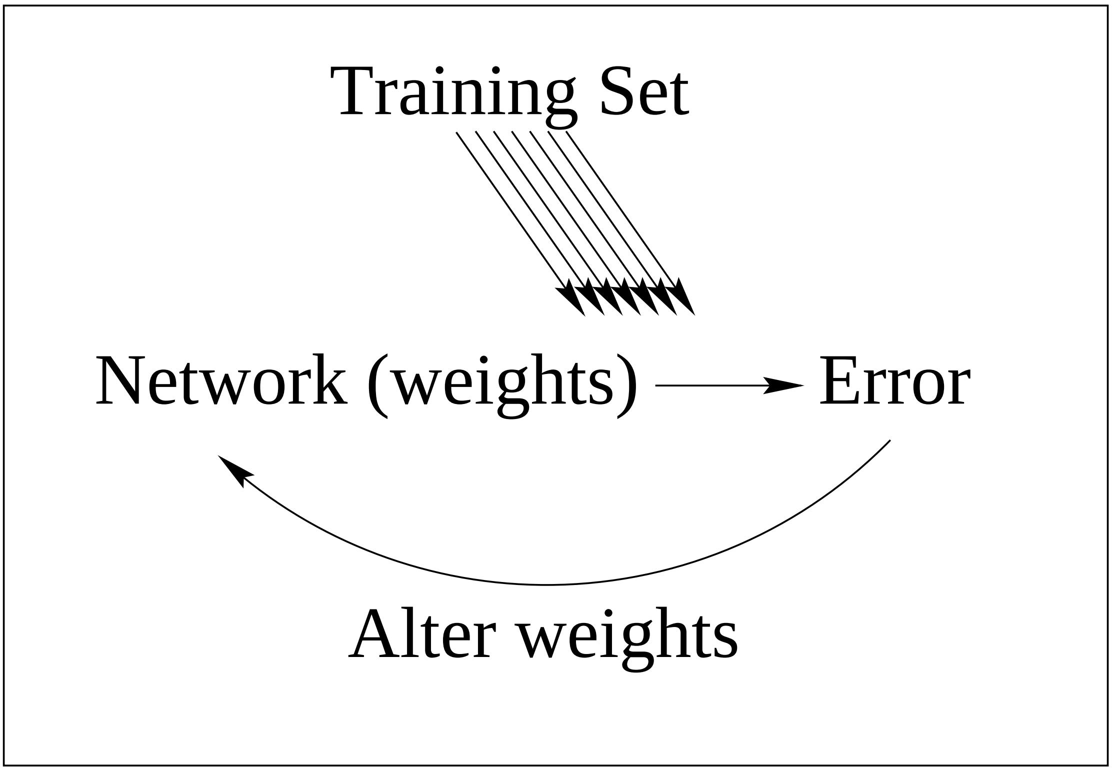


Figure 7.1: Training cycle of feed-forward, backprop network

- Many evaluations of error, but cannot run multiple evaluations of error in parallel throughout the code – in many places it is serial! Only error evaluations in the "genetic" part are parallel.
- Training set of example is small, but training sets get *much* larger. Training sets are static (fixed once at the beginning).
- Error cumulative!

This suggests as a solution:

- Send training set to nodes (once). Send weights to nodes each time error is needed. Split up application of network to training set among the nodes, cumulate resulting error and reassemble into total error.
- "Master-Slave" paradigm. Often good for beowulf, but beware accumulation of NEW serial time associated with bottlenecked IPC's...

Bottlenecks

Bottlenecks? What are those? Bottlenecks are by definition rate determining factors in code execution (serial or parallel). We need to be aware of various bottlenecks:

- CPU. The CPU itself is often the primary bottleneck. This is usually a Good Thing for a beowulf application, since CPU is what you get more of in a parallel system.
- Input/Output (I/O). The disk, the keyboard, video – all MUCH slower than processing itself (and probably serial, recall word processor).
- Memory. CPU speed has grown faster than memory speed can keep up. Which leads us to...
- Cache. A cache is a small block of "superfast" memory attached directly to the CPU. All sorts of potential bottlenecks (and optimizations) here.
- Kernel. Systems calls may be fast or slow or blocked (SMP).
- Network. In a beowulf, the network is the "interprocessor communications channel" (IPC). This is such an important and complex bottleneck that we consider it in detail later.
- These bottlenecks all interact, sometimes in surprising ways.

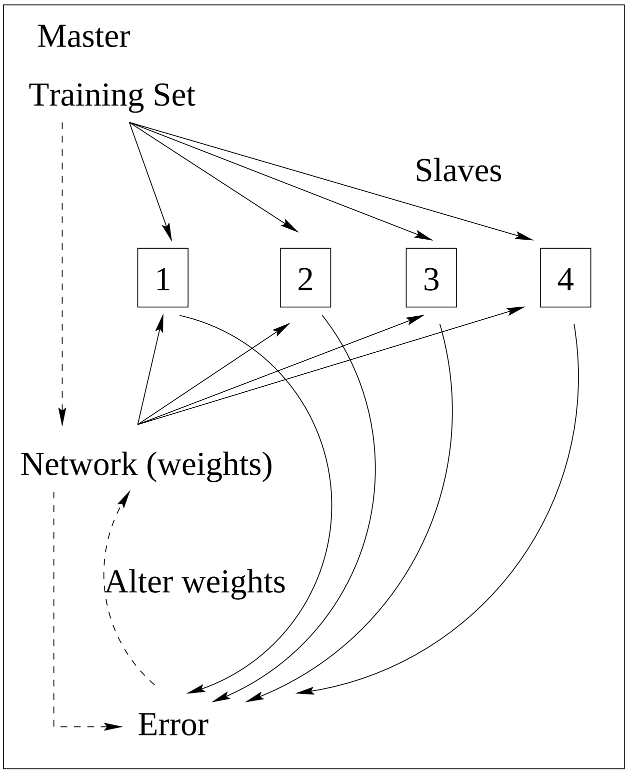


Figure 7.2: Training cycle of parallelized feed-forward, backprop network

T_p vs IPC Time T_i
The Second Step

Suppose that after profiling your task (like the discover example) appears suitable for parallelization. Are you done studying your code? Definitely not.

Look at the very rough schematic of our parallel neural training cycle.

Every solid arrow crudely represents an interprocessor communication cycle that takes *time* we represent T_i. In a master-slave paradigm this time adds to the *serial* time T_s. In a more symmetric communcations model, part of this time might itself be parallelized.

Parallelizing code *changes the serial and parallel fractions!*

This introduces new P variation into our earlier statement of Amdahl's Law. Call $T_{i,s}$ the serial IPC time per node and call $T_{i,p}$ the parallel IPC time (which still adds to the serial time). Then following Amalsi and Gottleib, we can crudely represent the modified "speed" as:

$$\frac{\mathrm{Work}}{(\mathrm{T_s+T_{i,p})+P*T_{i,s}+T_p/P}}$$

We see that $T_{i,s} \neq 0$ will ALWAYS prevent us from profitably reaching $P \to \infty$. Amalsi and Gottlieb write this modified version of Amdahl's Law for the special case of a Master-Slave algorithm as:

$$\frac{1}{S+(1-S)/P+P/r_1}$$

where they've introduced $r_1 = T_s/(P * T_{i,s})$, the ratio of compute time to serial communications time, per node.

Even THIS isn't pessimistic enough. It assumes "perfect" parallelizability. In real code, the parallel tasks may well need to be organized so that they complete in certain orders and parts are available where they are needed just when they are needed. Then there may be random delays on the nodes due to other things they are doing. All of this can further degrade performance.

Which leads us to...

Problem Granularity and Synchronicity
The Key Parameters of Beowulf Design

Let us define two general kinds of parallel subtasks:

- Coarse Grain subtasks are those where $T_p \gg T_i$ ($r_1 \gg 1$) for both serial and parallel IPC's. Life is good – computation dominates IPC's.
- Fine Grain subtasks are those where $T_p \sim T_i$ ($r_1 \sim \mathcal{O}(1)$), or worse. Too bad, IPC's dominate computation.

In addition, each of these subtasks may be *synchronous* where they all have to proceed together or *asynchronous* where they can proceed effectively independently. This gives us at least four (really more) descriptors like "coarse grained asynchronous code" or "fine grained, tightly coupled synchronous code".

The former is easy to program on nearly any beowulf (or cluster!). There are many valuable tasks in this category, which partly explains the popularity of beowulfs.

Coarse grained synchronous code can also run well on most simple beowulf designs, although it is harder to program and load balance efficiently. What about moderate to fine grain code, though?

The Good News – and the Bad News

The good news is that one can often find great profit using beowulfs to solve problems with moderate to fine granularity. Of course, one has to work harder to obtain such a benefit, both in programming and in the design of the beowulf! Still, for certain problem scales the cost/benefit advantage of a "high end" beowulf solution may be an order of magnitude greater than that of any competitive "big iron" supercomputer sold by a commercial vendor.

There are, however, limits. Those limits are time dependent (recall Moore's Law) but roughly fixed on the timescale of the beowulf design and purchase process. They are basically determined by what one can cost-effectively buy in M^2-COTS components.

For any given proposed beowulf architecture, if T_p is anywhere *close* to $T_{i,s}$, chances are good that your parallel efforts are doomed. When it takes longer to communicate what is required to take a parallel synchronous step than it does to take the step, parallelization yields a negative benefit unless calculation size is your goal.

Beowulf hardware and software engineering consists of MAKING YOUR PROBLEM *RELATIVELY* COARSE GRAINED ON THE (COTS) HARDWARE IN QUESTION. That is, keeping $T_{i,s}$ under control.

Estimating or Measuring Granularity

Estimating is difficult. Inputs include:

- "Bare" estimates of T_s and T_p (determined from gprof)..
- Raw network bandwidth (10 Mbps, 100 Mbps, 1000 Mbps) (test with netperf, ttcp).
- Raw network latency (extremely variable) (test with netperf).
- Contributions and tradeoffs galore. The protocol stack, the paradigm, hardware bottlenecks, the kernel, the interconnection structure, the attempted number of nodes – all nonlinearly interact to produce T_i and the modified T_s and T_p.

Experts rarely analyze beyond a certain point. They measure (or just know) the base numbers for various alternatives and then prototype instead. Or they ask on lists for experiences with similar problems. By far the safest and most successful approach is to build (or borrow) a *small* 3-4 node switched 100BT cluster (see recipe above) to prototype and profile your parallel code.

Remember that granularity is often something you can control (improve) by, for example, working on a bigger problem or buying a faster network. Whether or not this is sensible is an *economic* question.

Repeat Until Done

Back to the Example

In a moment, we will think about specific ways to improve granularity and come up with a *generalized* recipe for a beowulf or cluster that ought to be able to Get the Job Done. First, let's complete the "study the problem" section by showing the results of prototyping runs of the "splitup the error evaluation" algorithm for the neural net example at various granularities, on a switched 100BT network of 400 MHz PII nodes.

```
\# First round of timing results
\# Single processor on 300 MHz master ganesh, no PVM
0.880user 21.220sys 99.9%, 0ib 0ob 0tx 0da 0to 0swp 0:22.11
0.280user 21.760sys 100.0%, 0ib 0ob 0tx 0da 0to 0swp 0:22.04
\# Single processor on 400 MHz slave b4 using PVM
0.540user 11.280sys 31.3%, 0ib 0ob 0tx 0da 0to 0swp 0:37.65
0.700user 11.010sys 31.1%, 0ib 0ob 0tx 0da 0to 0swp 0:37.62
\# 2x400 MHz (b4, b9) with PVM
1.390user 14.530sys 38.3%, 0ib 0ob 0tx 0da 0to 0swp 0:41.48
\# 3x400 MHz (b4, b9, b11) with PVM
1.800user 18.050sys 46.5%, 0ib 0ob 0tx 0da 0to 0swp 0:42.60
```

This, of course, was terrible! The problem slowed down when we run it in parallel! Terrible or not, this is *typical* for "small" prototyping runs and we should have expected it.

Clean Up the Hacks

We made two changes in the code. First, we eliminated some debugging cruft in the slave code that was increasing the bottlenecked serial fraction. Second, originally we multicast the network but sent each host its slice boundaries serially. This, in retrospect, was stupid, as the communication was latency bounded, not bandwidth bounded (small messages nearly always are). Instead we multicast the entire slave slice assignments along with the weights and then awaited the slave results.

The results now:

```
\# Single processor on 300 MHz master ganesh, no PVM.  Guess not.
1.250user 20.630sys 99.9%, 0ib 0ob 0tx 0da 0to 0swp 0:21.90
\# Single processor on 400 MHz slave b4 using PVM.  Better.
0.350user 10.460sys 32.9%, 0ib 0ob 0tx 0da 0to 0swp 0:32.79
2.380user 8.410sys 32.5%, 0ib 0ob 0tx 0da 0to 0swp 0:33.11
\# 2x400 MHz (b4, b9) with PVM
2.260user 11.140sys 37.7%, 0ib 0ob 0tx 0da 0to 0swp 0:35.53
\# 3x400 MHz (b4, b9, b11) with PVM
1.630user 11.160sys 40.3%, 0ib 0ob 0tx 0da 0to 0swp 0:31.67
\# 4x400 MHz (b4, b9, b11, b12) with PVM
2.720user 14.720sys 42.9%, 0ib 0ob 0tx 0da 0to 0swp 0:40.61
```

Still no gain, but closer!

Crank Up the Granularity

Finally, we tried increasing the granularity a bit by using a bigger dataset. We thus used a 16 bit divide by sevens problem. Small as the increase was, it was big enough:

```
\# Single processor on 300 MHz master ganesh, no PVM.  Takes longer.
9.270user 207.020sys 99.9%, 0ib 0ob 0tx 0da 0to 0swp 3:36.32
\# Single processor on 400 MHz slave b4 using PVM.  Better.
4.380user 61.410sys 28.3%, 0ib 0ob 0tx 0da 0to 0swp 3:51.67
\# 2x400 MHz (b4, b9) with PVM.  At last a distinct benefit!
3.080user 71.420sys 51.1%, 0ib 0ob 0tx 0da 0to 0swp 2:25.73
\# 3x400 MHz (b4, b9, b11) with PVM. Still better.
1.270user 70.570sys 58.9%, 0ib 0ob 0tx 0da 0to 0swp 2:01.89
\# 4x400 MHz (b4, b9, b11, b12) with PVM. And peak.
6.000user 71.820sys 63.3%, 0ib 0ob 0tx 0da 0to 0swp 2:02.83
\# More processors would actually cost speedup at this granularity.
```

We're Home! A nice speedup, even for this SMALL (toy) problem. But why are we bothering?

Show me the Money...

We're bothering because predictive modeling is *valuable* and time is money. In an actual credit card cross-sell model built for a large North Carolina bank (with 132 distinct inputs – optimization in 132 dimensions with sparse data!), it took a full day and a half to run a single full network training cycle on a single processor PII at 450 MHz. This can be too long to drive a real-time direct phone campaign, and is annoyingly long from the point of view of tying up compute resources as well.

A smaller version of the same credit card model was also run with only 22 inputs. This model required over two hours to run on a 400 MHz PII. We benchmarked our new parallel neural network program on this smaller model to obtain the following:

```
# CCA with 22 inputs.  There are well over 4 million quadrants and only
# a few thousand members in the training set!  A truly complex problem.
# Time with just one serial host
442.560user 7618.620sys 99.9%, 0ib 0ob 0tx 0da 0to 0swp 2:14:26.12
# Time with two PVM hosts
112.840user 1999.970sys 37.4%, 0ib 0ob 0tx 0da 0to 0swp 1:34:06.02
# Time with five PVM hosts
95.030user 2361.560sys 60.0%, 0ib 0ob 0tx 0da 0to 0swp 1:08:11.86
```

Discover$_{\rm tm}$ Conclusions

The scaling of our preliminary parallelization is still worse than we might like, but the granularity is *still* a factor of 5 to 10 smaller than the real models we wish to apply it to. We expect to be able to obtain a maximum speedup of five or more with about eight Celeron nodes in actual application (that cost little more altogether than many of our single or dual CPU PII's did originally).

Finally, our profiling indicates that about 2/3 of the remaining serial code (the regression routine, part of the conjugate gradient cycle, and the genetic algorithm itself) can be parallelized as well. Using this parallelized network, we expect to be able to tackle bigger, more complex networks and still get excellent results.

This, in turn, will make both our clients money and (we hope) us money. Thar's Gold in Them Thar Hills (of the joint probability distribution being modeled, of course)...

At Last...How to Design a Beowulf

By this point, the answer should be obvious, which is why I saved it until now. AFTER one has finished studying the problem, or problems, one plans to run on the beowulf, the design parameters are real things that apply to the actual bottlenecks you encountered and parallel computation schema you expect to implement, not just things “rgb told me to use”. The following is a VERY ROUGH listing of SOME of the possible correspondances between problem and design solution:

Problem: Embarrassingly coarse grained problems; e.g. Monte Carlo simulations.

Solution: Anything at all. Typically CPU bound, r_1 all but infinite. I can get nearly perfect parallelization of my Monte Carlo code by walking between consoles of workstations, loading the program from a floppy, and coming back later to collect the results on the same floppy. Beowulf based on sneakernet, yeah! Of course, a network makes things easier and faster to manage...

Advise to builders: Focus on the CPU/memory cost/benefit peak and single system bottlenecks, not the network. Get a decent network though – these days switched 100 BT is sort of the lowest common denominator because it is so cheap. You might want to run your simulations in not-so-coarse grain mode one day. Also be aware that ordinary workstation clusters running linux can work on a problem with 98% of the CPU and still provide "instant" interactive response. A MAJOR REASON for businesses to consider linux clusters is that their entire office can "be" a parallel supercomputer even while the desktop units it's composed of enable folks to read mail and surf the web! No Microsoft product can even think of competing here.

Problem: Coarse grained problems (but not embarrassingly so) to medium grain problems; e.g. Monte Carlo problems where a lattice is split up across nodes, neural networks.

Solution: The "standard beowulf" recipe still holds IF latency isn't a problem. A switched 100 BT network of price/performance-optimal nodes is a good choice. Check carefully to ensure that cache size and memory bus are suitable on the nodes. Also, take more care that the network itself is decent – you do have to transmit a fair amount of data between nodes, but there are clever ways to synchronize all this. If bandwidth (not latency) becomes a problem, consider channel bonding several 100 BT connections through a suitable switch.

Advise to builders: Think about cost/benefit very carefully. There is no point in getting a lot more network than you need right now. It will be faster and cheaper next year if that's when you'll actually (maybe) need it. Get a cheap net and work up. Also do you really need 512 MB of node memory when your calculation only occupies 20 MB? Do you need a local disk? Is cache or cost a major factor? Are you really CPU bound and do you need very fast nodes (so Alpha's make sense)?

You are in the "sweet spot" of beowulf design where they are really immensely valuable but not too hard or expensive to make. Start small, prototype, scale up what works.

Problem: Medium to fine grained problems; e.g. molecular dynamics with long range forces, hydrodynamics calculations – examples abound. These are the problems that were once the sole domain of Big Iron "real" parallel supercomputers. No more.

Solution: Make the problem coarse grained, of course, by varying the design of the program and the beowulf until this can be achieved. As Walter Ligon (a luminary of the beowulf list) recently noted, a beowulf isn't really suited for fine grained code. Of course, *no* parallel computing environment is well-suited for fine grained code – the trick is to pick an environment where the code you want to run has an acceptable granularity. Your tools for achieving this are clever and wise programming, faster networks and possibly nodes, and increasing the problem size.

The "standard" solution for fine(r) grain code is to convert to Myrinet (or possibly gigabit ethernet as its latency problem is controlled). This can reduce your T_i by an order of magnitude if you are lucky, which will usually make a fine grained problem coarse enough to get decent gain with the number of processors once again. If your problem is (as is likely enough) ALSO memory bound (big matrices, for example), possessed of a large stride (ditto), and CPU bound, seriously consider the AlphaLinux+Myrinet solution described by Greg Lindahl (for example) or wait for the K7 or Merced. If it is just IPC bound, it may be enough to get a faster network without increasing CPU speed (and cost) significantly – diverting a larger fraction of one's resources to the network is the standard feature of dealing with finer problem granularities.

Advise to builders: Take the problem seriously. Get and read Almasi and Gottlieb or other related references on the theory and design of parallel code. There are clever tricks that can significantly improve the ratio of computation to communication and I've only scratched the surface of the theory. Don't be afraid to give up (for now). There are problems that it just isn't sensible to parallelize. Also don't be put off by a bad prototyping experience. As one ramps up the scale (and twiddles the design of the beowulf) one can often get dramatic improvements.

Summary

- Remember Amdahl's Law (and variants)
- Bottlenecks (serial and parallel)
- Make crude estimates of T_s, T_p, and $T_{i,s/p}$.
- Give up if T_p/T is too small to be worth it.
- Seek cheapest/simplest design for which T_p/T_i and AL predict decent speedup for a cost-effective value of P.
- Beware nonlinearities in general, P-dependent serial costs in T_i especially (common in master/slave) and remain aware of synchronization issues.

Conclusion

Beowulfs and linux clusters in general are an amazingly cost effective way to collect the cycles necessary to do large scale computing, *if* your problem has an appropriate granularity and parallelizable fraction. On the advanced end, they are rapidly approaching the efficiency of systems that cost ten or more times as much from commercial vendors. On the low end, they are bringing supercomputing "home" to elementary schools and even homes (this cluster lives in my "typical" home, for example).

There are clearly huge opportunities for making money by solving previously inaccessible problems using this technology, especially in business modeling and data mining. E pluribus penguin, and for Microsoft sic gloria transit mundi.

References

First, the beowulf and linux-smp lists. For years. Check the archives.

Second, "Highly Parallel Computing", by Almasi and Gottlieb.

Third, "How To Build a Beowulf", by Sterling, Becker, et. al.

Chapter 8

Specific Parallel Models

8.1 Embarrassingly Parallel Computations

We've already talked some about EPC's, but as they are the bread and butter of cluster computation we'll consider them in detail here. If your computational work tends to consist of running a series of more or less independent jobs on your computer, perhaps varying the parameters (or perhaps not), then you are an obvious candidate for a compute cluster. You are the most fortunate of souls, as well, because in most cases you don't much care about the design details of the cluster.

There are many useful tasks that fall into this general category, but perhaps the cleanest example of an EPC task is Monte Carlo calculations and other forms of statistical simulation. This is near and dear to my own heart as it is what I do as a physicist[1].

8.1.1 The Network is the Computer: MOSIX

8.1.2 Batch Systems with a Heart: Condor

8.1.3 Master-Slave Calculations

A classic example of this is the version of Amdahl's Law for the special case of a master-slave algorithm, where one "master" node sends out the task(s) to P "slave" nodes[2]. This is in the class of "simple" parallel tasks, so the formula above works, often with just one parallelizable subtask that the "master" node splits up and sends to the "slave" nodes to do. If you sat at the front of a room filled with 10 friends with a stack of 100 model airplanes to build, a master-slave approach is to walk around handing out a kit to each and then kick back, tapping your feet for twenty minutes until they all finish. Then you walk around

[1] In case you were wondering how a physicist came to be writing this book, I put the story – which is not without its edifying points – in an Appendix at the end.

[2] It's not what you were thinking it was at all, was it? Shame on you.

and pick up the finished airplanes and give them the next kit. You might then even do some work on them yourself (put them in a box to be shipped to model-airplaneless children, for example, or throw them out a window to see if they fly) while they work on the next ones, repeat until done[3].

From this we see that master-slave calculations (MSC) have an intimate relationship with embarrassingly course grained calculations (ECGC) – one way to implement nearly any ECGC is as a MSC. So to speak. However, not all MSC are ECG. To see this, we have to learn what the word's "coarse grained" (or medium grained of fine grained or granularity in general) mean in the general context of parallel computing. We'll do this later.

Amdahl's Law becomes:

$$\frac{R(P)}{R(1)} \leq \frac{1}{S + (1 - S)/P + P/r_1}. \tag{8.1}$$

In this, S is the serial fraction of the code (the time you spend piling all the boxes up on a table before beginning), $(1 - S)$ is the parallel fraction (building the airplanes and throwing them out the window), and $r_1 = T_s/(P * T_{i,sp})$, the ratio of serial compute time to serial communications time, per node (the

Even THIS isn't pessimistic enough. It assumes "perfect" parallelizability. In real code, the parallel tasks may well need to be organized so that they complete in certain orders and parts are available where they are needed just when they are needed. Then there may be random delays on the nodes due to other things they are doing. All of this can further degrade performance.

Which leads us to...

8.2 Lattice Models

8.3 Long Range Models

[3] At the end of much more of this, you won't HAVE any friends. Who wants to be a slave?

Part III

Beowulf Hardware

Chapter 9

Node Hardware

9.1 Rates, Latencies and Bandwidths

In order to achieve the best scaling behavior, we can see from the previous chapter that we want to maximize the parallel fraction of a program (the part that can be split up) and minimize the serial fraction (which cannot). We also want to maximize the time spend doing work in parallel on each node and minimize the time required to communicate between nodes. We want to avoid wasting time by having some nodes sit idle waiting for other nodes to finish.

However, we must be cautious and clearly define our *real* goals as in general they aren't to "achieve the best scaling behavior" (unless one is a computer scientist studying the abstract problem, of course). More commonly in application, they are to "get the most work done in the least amount of time given a fixed budget". When *economic constraints* appear in the picture one has to carefully consider trade-offs between the computational speed of the nodes, the speed and latency of the network, the size of memory and its speed and latency, and the size, speed and latency of any hard storage subsystem that might be required. Virtually any conceivable combination of system and network speed can turn out to be cost-benefit optimal and get the most work done for a given budget and parallel task.

As the discussion proceeds, it will become clear why successful beowulf design focusses on the *problem* as much as it does on the hardware. One perfectly reasonable conclusion a reader can draw from this chapter is that *understanding* the nuances of computer hardware and their effect on program speed is ludicrously difficult and that only individuals with some sort of obsessive-compulsive personality disorder would ever try it. It is so much simpler to just *measure* the performance of any given hardware platform on *your* program.

When you achieve this Satori, Bravo! However, be warned that the wisest course is to *both* measure performance *and* understand at least a bit about how that measured performance is likely to vary when you vary things like the clock speed of your CPU, the CPU's manufacturer, the kind and speed of the memory subsystem, and the network.

The variations can be profound. As we'll see, when we *double* the size of (say, vectors being multiplied within) a program it can take *twelve or more* times as long to complete for certain ranges of sizes. You could naively make your measurement where performance is great, expecting it to remain great in production for much larger vectors or matrices. You could then expend large sums of money buying nodes, fail miserably to get the work accomplished that you expected to accomplish for that sum, and (fill in your own favorite personal disaster) get fired, not get tenure, lose your grant, go broke, lose the respect of your children and pets. Alternatively, you could make your measurement for *large* matrices, assume that fast memory systems are essential and spend a great deal for a relative few of them, only to find that by the time the problem is split up it would run just as fast on nodes with slower memory that cost 1/10 as much (so you could have bought 10x as many).

This isn't as hard to understand is it may now seem. I'll try to explain how and why this can occur and illustrate it with enough pictures and graphs and examples that it becomes clear. Once you understand what this chapter has to offer, you'll understand *how* to study your problem in a way that is unlikely to produce embarrassing and expensive mistakes in your final beowulf design.

9.1.1 Microbenchmarking Tools

Finding the truly optimum design can be difficult. In some cases the *only* way to determine a program's performance on a given hardware and software platform (or beowulf design) is to do a lot of prototyping and benchmarking of the program itself. From this one can generally determine the best design empirically (where hopefully one has enough funding in these cases to fund the prototyping and then scale the successful design up into the production beowulf). This is almost always the *best* thing to do, if one can afford it.

However, *even* if you are able to prototype and benchmark your actual application, the design process is significantly easier if one possesses a detailed and quantitative knowledge of various microscopic *rates, latencies,* and *bandwidths* and how they depend *nonlinearly* on certain system and program parameters and features. Let's begin by understanding just what these things are.

- A *rate* is a given number of operations per unit time, for example, the number of double precision multiplications a CPU can execute per second. We might like to know the "maximum" rate a CPU can execute floating point instructions under ideal circumstances. We might be even more interested in how the "real world" floating point rate depends on (for example) the size and locality of the memory references being operated upon.

- A *latency* is is the time the CPU (or other subsystem) has to *wait* for a resource or service to become available after it is requested and has units of an inverse rate – milliseconds per disk seek, for example. A latency isn't necessarily the inverse of a rate, however, because the latency often is very different for an isolated request and a streaming series of identical requests.

- A *bandwidth* is a special case of a rate. It measures "information per unit time" being delivered between subsystems (for example between memory and the CPU). Information in the context of computers is typically data or code organized as a byte stream, so a typical unit of bandwidth might be megabytes per second.

Latency is *very* important to understand and quantify as in many cases our nodes will be literally sitting there and twiddling their thumbs waiting for a resource. Latencies may be the *dominant* contribution to the communications times in our performance equations above. Also (as noted) rates are often the inverse of some latency. One can equally well talk about the rate that a CPU executes floating point instructions or the latency (the time) between successive instructions which is its inverse. In other cases such as the network, memory, or disk, latency is just one factor that contributes to overall rates of streaming data transfer. In general a large latency translates into a low rate (for the same resource) for a small or isolated request.

Clearly these rates, latencies and bandwidths are important determinants of program performance even for single threaded programs running on a single computer. Taking advantage of the nonlinearities (or avoiding their *dis*advantages can result in dramatic improvements in performance, as the ATLAS (Automatically Tuned Linear Algebra System) [ATLAS] project has recently made clear. By adjusting both algorithm and blocksize to maximally exploit the empirical speed characteristics of the CPU in interaction with the various memory subsystems, ATLAS achieves a factor of two or more improvement in the excution speed of a number of common linear operations. Intelligent and integrated beowulf design can similarly produce startling improvements in both cost-benefit and raw performance for certain tasks.

It would be very useful to have automatically available all of the basic rates that might be useful for automatically tuning program and beowulf design. At this time there is no daemon or kernel module that can provide this empirically determined and standardized information to a compiled library. As a consequence, the ATLAS library build (which must measure the key parameters in place) is so complex that it can take hours to build on a fast system.

There do exist various standalone (open source) microbenchmarking tools that measure a large number of the things one might need to measure to guide thoughtful design. Unfortunately, many of these tools measure only isolated performance characteristics, and as we will see below, isolated numbers are not always useful. However, one toolset has emerged that by design contains (or will soon contain) a *full suite* of the elementary tools for measuring precisely the rates, latencies, and bandwidths that we are most interested in, using a common and thoroughly tested timing harness. This tool is not complete[1] but

[1]More time was spent by the author of this paper working on and with the tool than on

it has the promise of becoming *the* fundamental toolset to support systems engineering and cluster design. It is Larry McVoy and Carl Staelin's "lmbench" toolset[lmbench].

There are two areas where the alpha version 2 of this toolset used in this paper was still missing tools to measure network throughput and raw "numerical" CPU performance (although many of the missing features and more have recently been added to lmbench by Carl Staelin after some gentle pestering). The well-known netperf (version 2.1, patch level 3) [netperf] and a privately written tool [cpu-rate] were used for this in the meantime.

All of the tools that will be discussed are open source in the sense that their source can be readily obtained on the network and that no royalties are charged for its use. The lmbench suite, however, has a general use license that is slightly more restricted than the usual Gnu Public License (GPL) as described below.

In the next subsections the results of applying these tools to measure system performance in my small personal beowulf cluster[Eden] will be presented. This cluster is moderately heterogeneous and functions in part as a laboratory for beowulf development. A startlingly complete and clear profile of system performance and its dependence on things like code size and structure will emerge.

9.1.2 Lmbench Results

In order to *publish* lmbench results in a public forum, the lmbench license *requires* that the benchmark code must be compiled with a "standard" level of optimization (-O only) and that *all* the results produced by the lmbench suite must be published. These two rules together ensure that the results produced compare as fairly as possible apples to apples when considering multiple platforms, and prevents vendors or overzealous computer scientists from seeking "magic" combinations of optimizations that improve one result (which they then selectively publish) at the expense of others.

Accordingly, on the following page is a full set of lmbench results generated for "lucifer", the primary server node for my home (primarily development) beowulf [Eden]. The mean values and error estimates were generated from averaging ten independent runs of the full benchmark. lucifer is a 466 MHz *dual* Celeron system, permitting it to function (in principle) simultaneously as a master node and as a participant node. The cpu-rate results are also included on this page for completeness although they may be superseded by Carl Staelin's superior hardware instruction latency measures in the future.

lmbench clearly produces an *extremely detailed* picture of microscopic systems performance. Many of these numbers are of obvious interest to beowulf designers and have indeed been discussed (in many cases without a sound quantitative basis) on the beowulf list [beowulf]. We must focus in order to conduct a sane discussion in the allotted space. In the following subsections on we will consider the network, the memory, and the cpu-rates as primary contributors to beowulf and parallel code design.

---

the paper:-)

HOST	lucifer
CPU	Celeron (Mendocino) (x2)
CPU Family	i686
MHz	467
L1 Cache Size	16 KB (code)/16 KB (data)
L2 Cache Size	128 KB
Motherboard	Abit BP6
Memory	128 MB of PC100 SDRAM
OS Kernel	Linux 2.2.14-5.0smp
Network (100BT)	Lite-On 82c168 PNIC rev 32
Network Switch	Netgear FS108

Table 9.1: Lucifer System Description

null call	0.696 ± 0.006
null I/O	1.110 ± 0.005
stat	3.794 ± 0.032
open/close	5.547 ± 0.054
select	44.7 ± 0.82
signal install	1.971 ± 0.006
signal catch	3.981 ± 0.002
fork proc	634.4 ± 28.82
exec proc	2755.5 ± 10.34
shell proc	10569.0 ± 46.92

Table 9.2: lmbench latencies for selected processor/process activities. The values are all times in microseconds averaged over ten independent runs (with error estimates provided by an unbiased standard deviation), so "smaller is better".

2p/0K	1.91 ± 0.036
2p/16K	14.12 ± 0.724
2p/64K	144.67 ± 9.868
8p/0K	3.30 ± 1.224
8p/16K	48.45 ± 1.224
8p/64K	201.23 ± 2.486
16p/0K	6.26 ± 0.159
16p/16K	63.66 ± 0.779
16p/64K	211.38 ± 5.567

Table 9.3: Lmbench latencies for context switches, in microseconds (smaller is better).

pipe	10.62 ± 0.069
AF UNIX	33.74 ± 3.398
UDP	55.13 ± 3.080
TCP	127.71 ± 5.428
TCP Connect	265.44 ± 7.372
RPC/UDP	140.06 ± 7.220
RPC/TCP	185.30 ± 7.936

Table 9.4: Lmbench *local* communication latencies, in microseconds (smaller is better).

UDP	164.91 ± 2.787
TCP	187.92 ± 9.357
TCP Connect	312.19 ± 3.587
RPC/UDP	210.65 ± 3.021
RPC/TCP	257.44 ± 4.828

Table 9.5: Lmbench *network* communication latencies, in microseconds (smaller is better).

L1 Cache	6.00 ± 0.000
L2 Cache	112.40 ± 7.618
Main mem	187.10 ± 1.312

Table 9.6: Lmbench *memory* latencies in nanoseconds (smaller is better). Also see graphs for more complete picture.

pipe	290.17 ± 11.881
AF UNIX	64.44 ± 3.133
TCP	31.70 ± 0.663
UDP	(not available)
bcopy (libc)	79.51 ± 0.782
bcopy (hand)	72.93 ± 0.617
mem read	302.79 ± 3.054
mem write	97.92 ± 0.787

Table 9.7: Lmbench *local* communication bandwidths, in 10^6 bytes/second (bigger is better).

TCP	11.21 ± 0.018
UDP	(not available)

Table 9.8: Lmbench *network* communication bandwidths, in 10^6 bytes/second (bigger is better).

Single precision	289.10 ± 1.394
Double precision	299.09 ± 2.295

Table 9.9: CPU-rates in BOGOMFLOPS – 10^6 simple arithmetic operations/second, in L1 cache (bigger is better). Also see graph for out-of-cache performance.

These are not at all independent. The rate at which the system does floating point arithmetic on streaming vectors of numbers is very strongly determined by the relative size of the L1 and L2 cache and the size of the vector(s) in question. Significant (and somewhat unexpected) structure is also revealed in network performance as a function of packet size, which suggests "interesting" interactions between the network, the memory subsystem, and the operating system that are worthy of further study.

9.1.3 Netperf Results

Netperf is a venerable and well-written tool for measuring a variety of critical measures of network performance. Some of its features are still not duplicated in the lmbench 2 suite; in particular the ability to completely control variables such as overall message block size and packet payload size.

A naive use of netperf might be to just call

```
netperf -H targethost
```

to get a quick and dirty measurement of TCP stream bandwidth to a given target. However, as the lmbench TCP latency shows (see table 5), it takes some 150-200 microseconds to transmit a one-byte TCP packet message (on lucifer) or at most 5000-7000 packets can be sent per second. For small packets this results in far less than the "wirespeed dominated" bandwidth – the actual bandwidth observed for small messages is dominated by *latency*.

For each message sent, the time required goes directly into an IPC time like T_{is}. In the minimum 200 microseconds that are lost, the CPU could have done tens of thousands of floating point operations! This is why network latency is an extremely important parameter in beowulf design.

Bandwidth is also important – sometimes one has only a single message to send between processors, but it is a large one and takes much more than the 200 microseconds latency penalty to send. As message sizes get bigger the system uses more and more of the *total* available bandwidth and is less affected by latency. Eventually throughput saturates at some maximum value that depends on many variables. Rather than try to understand them all, it is is easier (and more accurate) to determine (maximum) bandwidth as a function of message size by direct measurement.

Both netperf and bw_tcp in lmbench allow one to directly select the message size (in bytes) to make a measurements of streaming TCP throughput. With a simple perl script one can generate a fine-grained plot of overall performance as a function of packet size. This has been done for a 100BT connection between

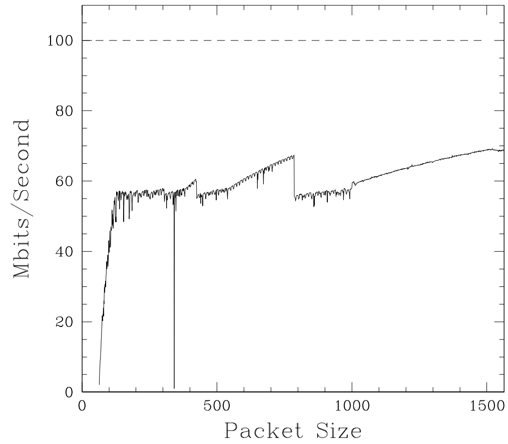


Figure 9.1: TCP Stream (netperf) measurements of bandwidth as a function of packet size between lucifer and eve.

lucifer and "eve" (a reasonably similar host on the same switch) as a function of packet size. These results are shown in figure 9.1.

Figure 9.1 reveals a number of surprising and even disappointing features. Bandwidth starts out small at a message size of one byte (and a packet size of 64 bytes, including the header) and rapidly grows roughly linearly at first as one expects in the latency-dominated regime where the number of packets per second is constant but the size of the packets is increasing. However, the bandwidth appears to *discontinuously* saturate at around 55 Mbps for packet sizes around 130 bytes long or longer. There is also considerable (unexpected) structure even in the saturation regime with sharp packet size thresholds. The same sort of behavior (with somewhat different structure and a bit better asymptotic large packet performance) appears when bw_tcp is used to perform the same measurement. We see that the single lmbench result of a somewhat low but relatively normal 11.2 MBps (90 Mbps) for large packets in table 8 hides a wealth of detail and potential IPC problems, although this single measure is all that would typically be published to someone seeking to build a beowulf using a given card and switch combination.

9.1.4 CPU Results

The CPU numerical performance is one of the most difficult components to precisely quantify. On the one hand, peak numerical performance is a measure always published by the vendor. On the other hand, this peak is basically never seen in practice and is routinely discounted.

CPU performance is known to be heavily dependent on just what the CPU does, the order in which it does it, the size and structure and bandwidths and latencies of its various memory subsystems including L1 and L2 caches, and the way the operating system manages cached pages. This dependence is extremely complex and studying one measure of performance for a particular set of parameters is not very illuminating if not misleading. In order to get any kind of feel at all for real world numerical performance, floating point instruction rates have to be determined for whole sweeps of e.g. accessed vector memory lengths.

What this boils down to is that there is very little numerical code that is truly "typical" and that it can be quite difficult to assign a single rate to floating point operations like addition, subtraction, multiplication, and division that might not be off by a factor of five or more relative to the rate that these operations are performed in *your* code. This translates into large uncertainties and variability of, for example, T_p with parallel program scale and design.

Still, it is unquestionably true that a detailed knowledge of the "MFLOPS" (millions of floating point operation per second) that can be performed in an inner loop of a calculation is important to code and beowulf design. Because of the high dimensionality of the variables upon which the rate depends (and the fact that we perforce must project onto a subspace of those variables to get any kind of performance picture at all) the resulting rate is somewhat bogus but not without it uses, *provided* that the tool used to generate it permits the exploration of at least a few of the relevant dimensions that affect numerical performance. Perhaps the most important of these are the various memory subsystems.

To explore raw numerical performance the cpu-rate benchmark is used [cpu-rate]. This benchmark times a simple arithmetic loop over a vector of a given input length, correcting for the time required to execute the empty loop alone. The operations it executes are:

```
x[i] = (1.0 + x[i])*(1.5 - x[i])/x[i];
```

where x[i] is initialized to be 1.0 and should end up equal to 1.0 (within any system roundoff error) afterwards as well.

Each execution of this line counts as "four floating point operations" (one of each type, where x[i] might be single or double precision) and by counting and timing one can convert this into FLOPS. As noted, the FLOPS it returns are somewhat bogus – they average over all four arithmetic operations (which may have very different rates), they contain a small amount of addressing arithmetic (to access the x[i] in the first place) that is ignored, they execute in a given order which may or many not accidentally benefit from floating point instruction pipelining in a given CPU, they presuming streaming access of the operational vector.

Still, this is more or less what what I think "most people" would mean when they ask how fast a system can do floating point arithmetic in the context of a loop over a vector. We'll remind ourselves that the results are bogus by labeling

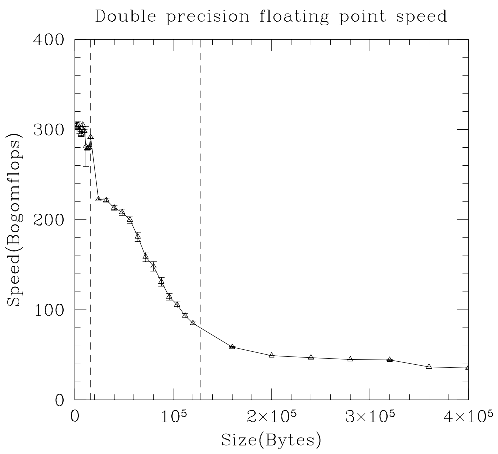


Figure 9.2: Double precision floating point operations per second as a function of vector length (in bytes). All points average 100 independent runs. The dashed lines indicate the locations of the L1 and L2 cache boundaries.

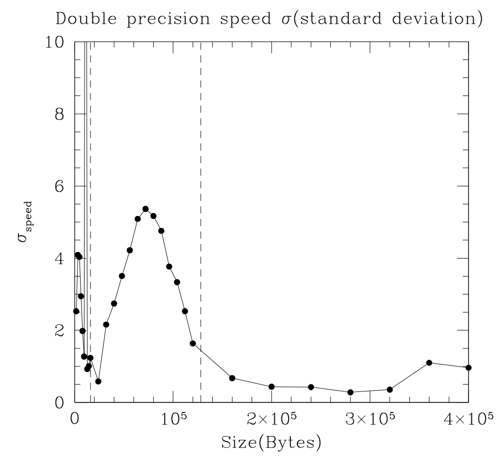


Figure 9.3: The standard deviation (error) associated with figure 9.2.

them "BOGOflops".

These rates will be *largest* when both the loop itself and the data it is working on are already "on the CPU" in registers, but for most practical purposes this rarely occurs in a core loop in compiled code that isn't hand built and tuned. The fastest rates one is likely to see in real life occur when the data (and hopefully the code) live in L1 cache, just outside the CPU registers. lmbench contains tests which determine at least the size of the L1 data cache size and its latency. In the case of lucifer, the L1 size is known to be 16 KB and its latency is found by lmbench to be 6 nanoseconds (or roughly 2-3 CPU clocks).

However, compiled code will *rarely* will fit into such a small cache unless it is specially written to do so. In any event we'd like to see what happens to the floating point speed as the length of the x[i] vector is systematically increased. Note that this measurement *combines* the raw numerical rate on the CPU with the effective rate that results when accounting for all the various latencies and bandwidths of the memory subsystem. Such a sequence of speeds as a function of vector lengths is graphed in figure 9.2.

This figure clearly shows that double precision floating point rates vary by *almost an order of magnitude* as the vector being operated on stretches from wholly within the L1 cache to several times the size of the L2 cache. Note also that the access pattern associated with the vector arithmetic is the *most favorable* one for efficient cache operation – sequential access of each vector element in turn. The factor of about *seven* difference in the execution speeds as the size of this vector is varied has profound implications for both serial code design and parallel code design. For example, the whole purpose of the ATLAS project [ATLAS] is to exploit the tremendous speed differential revealed in the figure by optimally blocking problems into in-cache loops when doing linear algebra operations numerically.

There is one more interesting feature that can be observed in this result. Because linux on Intel lacks page coloring, there is a large variability of numerical speeds observed between runs at a given vector size depending on just what pages happen to be loaded into cache when the run begins. In figure 9.3 the *variability* (standard deviation) of a large number (100) of independent runs of the cpu-rate benchmark is plotted as a function of vector size. One can easily pick out the the L1 and L2 cache boundaries as they neatly bracket the smooth peak visible in this figure. Although the L1 cache boundary is simple to determine directly from tests of memory speed, the L2 cache boundary has proven difficult to directly observe in benchmarks *because* of this variability. This is a new and somewhat exciting result – L2 boundaries can be revealed by a "susceptibility" of the underlying rate.

9.2 Conclusions

We now have many of the ingredients needed to determine how well or poorly lucifer (and its similar single-Celeron nodes, adam, eve, and abel) might perform on a simple parallel task. We also have a wealth of information to help us tune the task on *each* host to both balance the loads and to take optimal advantage of various system performance determinants such as the L1 and L2 cache boundaries and the relatively poor (or at least inconsistent) network. These numbers, along with a certain amount of judicious task profiling (for a description of the use of profiling in parallelizing a beowulf application see [profiling]) can in turn be used to determine the parameters that describe a given task like T_s, T_p, T_{is} and T_{ip}.

In addition, we have scaling curves that indicate the kind of parallel speedup we can expect to obtain for the task on the hardware we've microbenchmark-measured, and by comparing the appropriate microbenchmark numbers we *might* even be able to make a reliable guess at what the numbers and scaling would be on related but slightly different hardware (for example on a 300 MHz Celeron node instead of a 466 MHz Celeron node).

With these tools and the results they return, one can at least imagine being able to scientifically:

- develop a parallel program to run efficiently on a given beowulf
- tune an existing program on a given beowulf by considering for example bottlenecks and program scale
- develop a beowulf to run a given parallel program efficiently
- tune an existing beowulf to yield improved performance on a given program, or
- simultaneously develop, improve, and tune a *matched* beowulf design and parallel program together

even if one isn't initially a true expert in beowulf or general systems performance tuning. Furthermore, by using the *same* tools across a wide range of candidate platforms and publishing the comparative results, it may eventually become possible to do the all important optimization of *cost-benefit* that is really the fundamental motivation for using a beowulf design in the first place.

It is the hope of the author that in the near future the lmbench suite develops into a more or less standard microbenchmarking tool that can be used, along with a basic knowledge of parallel scaling theory, to identify and aggressively attack the critical bottlenecks that all too often appear in beowulf design and operation. An additional, equally interesting possibility would be to transform it into a daemon or kernel module that periodically runs on all systems and provides a standard matrix of performance measurements available from simple systems calls or via a /proc structure. This, in turn, would facilitate many, many aspects of the job of *dynamically* maximizing beowulf or general systems performance in the spirit of ATLAS but without the need to rebuild a program.

Chapter 10

Network Hardware

This chapter is devoted to perhaps the most important part of cluster design – the network. After all, nodes are relatively simple – you get what you pay for, you pay for what you get. For the most part, within a processor family, your serial task performance scales in fairly obvious ways with cpu clock, memory amount, type, speed, and so forth.

Not so with networking. Networks are intrinsically complex. In addition to the barebones concepts of latency and bandwidth we've already covered, issues like *topology*, *probability*, *task organization*, and various pieces of deep hardware-level magic come into play. Oh, and let's not forget the kernel, the device driver that interfaces device with the kernel, the networking stack that lives on *top* of the hardware device and its kernel device driver, the API between the driver and/or its networking stack, and of course your application. With maybe a layer or two more in there – no kidding.

Networks are *so* complex that I've found writing this chapter to be somewhat daunting and have hence postponed it again and again. They are also *so* rapidly varying that no matter what I write, the part at the bleeding edge will be obsolete (or at least, no longer bleeding edge and probably partly wrong) almost as fast as I write it and put it out there on the web.

Still, it's jobs that never get started that takes longest to finish, as Sam Gamgee's gaffer would say. So let's have at it. If you read this chapter and find that it is still incomplete, a) no surprise, it will take me a long time to write even a decently complete first draft; and b) feel free to bug me to finish it.

By the same token, if you really *understand* networking already and find something in this chapter that is egregiously in error (or for that matter, very subtley in error) you should feel free to correct me, if necessary with a whomp upside the head. I'm doing my best here, but some of these networks are expensive and until I either really need them or the vendors decide that they just *have* to loan me or give me a half-dozen cards worth to test and write about, I'm going to be writing at least partly on the basis of the vendors' published specifications, what I've gleaned from the beowulf list, what friends of mine who *do* run the networks have told me.

Finally, if you are a *real* expert in one of the high end networks and find my articles below to be hopelessly incompetent, well, remember that this is an *open source, open license* publication, and that I would cherish contributions from real experts. I'll even leave your very own name at the top of the section title, and ensure that it appears with the chapter in the TOC as well, so you get proper blame – uh, I mean credit – instead of me. High end networking companies, this goes for you as well – feel free to write your OWN (non-marketing-hype) description of your networking including a cost-benefit analysis and I'll cheerfully include it as a subsection – look, ma, free marketing, right where it does the most good!

The major editorial point being, the readers of this online book want useful, informed information about your products so they can make intelligent, cost-beneficial decisions about spending their money. You'd *like* for them to have all this information about *your* product so that they'll *choose* to buy it. Fine, we can work together on that basis, as long as you don't disrespect other's products or get into marketspeak. The readers of this document are all very likely to be technically competent shoppers and will want a technical and economic presentation, complete with at least ballpark prices.

Now, on to the meat of the matter. I'm going to try to organize this document in the following way. First, I'll present a very modest review of the basic concepts of networking, such as the ISO/OSI layers, the concept and general structure of a "packet", a bit of discussion of latency and bandwidth again (sorry, but this is a key context and requires it), and anything else likely to apply to "all networks". As a subsection, I'll present an equally compressed view of TCP/IP as one particular, important implementation of the network and transport layers. This won't be anywhere near enough to teach you to manage a TCP/IP network, but should give you a working knowledge of its basic concepts.

Then we'll run a set of sections on particular physical networks: Ethernet, SCI, Myrinet, and possibly more exotic networks (e.g. HIPPI) as I have the time and patience to figure them out or someone knowledgeable volunteers to write for me.

To permit me to reissue a book snapshot with this chapter *finally* not empty before I finish it, I'm going to cheat. A lot of what I'm going to put in this chapter comes from resources that I've either patiently collected over the last seventeen or so years or from resources that are readily available online. In fact, since most of the resources I've collected are ones that *I* make available online, one could say that *all* of it can be found on the web – somewhere (except where a copyright problem exists that might preclude republication). So for starters, every currently planned chapter will contain at the very least a list of web-resources you can click through – a reference to the vendor's website, for example.

That way, even if I'm still relatively ignorant of the network in question or am a world's-greatest-expert but just haven't had time to write the document (and which is which, you wonder, heh, heh) you'll be able to make some progress toward the all-important decision: what is the right network for my cluster?

Just to preview a part of the answer before we get started – it will almost certainly be TCP/IP on top of switched 100BT ethernet *and* (possibly) one of the high end, expensive the networks, depending on what you're planning to do. Switched 100BT has gone from nonexistent to expensive (as in tens of thousands of dollars) to quite cheap indeed in the years I've been doing cluster computing, and at this point it is *so* cheap, *so* ubiquitous, and *so* adequate for routine networking chores that it is hard to imagine a network without it. Perhaps in a few years it will be superceded by 1000BT ethernet as it once superceded 10BT ethernet, but in the meantime it is all but universal.

10.1 Basic Networking 101

Until this section gets filled in, here are some online resources:

- Charles L. Hedrick's

  Introduction to the Administration of a Local Area Network

- Charles L. Hedrick's

  Introduction to the Internet Protocols

- W. Richard Stevens'
  Personal Website.

  The late Dr. Stevens was *the* man where TCP/IP networking is concerned. His books are legendary. They are listed and linked to this website, although there are also various online resources and recipes here, primarily for the programmer.

10.1.1 Networking Concepts

10.1.2 TCP/IP

10.2 Ethernet

Until this section gets filled in, here are some online resources:

- Charles Spurgeon's Ethernet Site

10.2.1 10 Mbps Ethernet

10.2.2 100 Mbps Ethernet

10.2.3 1000 Mbps Ethernet

10.3 The Dolphin Serial Channel Interconnect

Until this section gets filled in, here are some online resources:

- `<a href="http://www.dolphinics.com/index.html">Dolphin Serial Channel Interconnect</a>`

10.4 Myrinet

Until this section gets filled in, here are some online resources:

- `<a href="http://www.myri.com/">Myricom Home Page</a>`

Part IV

Building a Beowulf

Chapter 11

Building and Maintaining a Beowulf

One question that is commonly enough asked on the beowulf list is "How hard is it to build or care for a beowulf?"

Mind you, it is quite possible to go into beowulfery with no more than a limited understanding of networking, a handful of machines (or better, a pocketful of money) and a willingness to learn, and over the years I've watched and sometimes helped as many groups and individuals (including myself) in many places went from a state of near-total ignorance to a fair degree of expertise on little more than guts and effort.

However, this sort of school is the school of hard (and expensive!) knocks; one *ought* to be able to do better and not make the same mistakes and reinvent the same wheels over and over again, and this book is an effort to smooth the way so that you can.

One place that this question is often asked is in the context of trying to figure out the human costs of beowulf construction or maintenance, especially if you're first cluster will be a big one and has to be right the first time. After all, building a cluster of more than 16 or so nodes is an increasingly serious proposition. It may well be that beowulfs are ten times cheaper than a piece of "big iron" of equivalent power (per unit of aggregate compute power by some measure), but what if it costs ten times as much in human labor to build or run? What if it uses more power or cooling? What if it needs more expensive physical infrastructure of any sort?

These are all *very valid* concerns, especially in a shop with limited human resources or with little linux expertise or limited space, cooling, power. Building a cluster with four nodes, eight nodes, perhaps even sixteen nodes can often be done so cheaply that it seems "free" because the *opportunity cost* for the resources required are so minimal and the benefits so much greater than the costs. Building a cluster of 256 nodes without thinking *hard* about cost issues, infrastructure, and cost-benefit analysis is very likely to have a very sad outcome, the *least* of which is that the person responsible will likely lose their job.

If that person (who will be responsible) is *you*, then by all means read on. I cannot guarantee that the following sections will keep you out of the unemployment line, but I'll do my best.

11.1 Physical Infrastructure

OK, so you've worked your way through the first umpty-chapters of this book (and possibly the Sterling, Salmon, Becker and Savarese book, and the HOWTO's, and the FAQ) and have picked a beowulf architecture. You've probably priced it out, as well, as cost-benefit was undoubtedly an important part of your selection criteria. You are probably *almost* ready to order the parts...

First, however, there are a few things you might *not* have thought about yet that you definitely need to consider[1]. Let's arrange them in a bulleted checklist (to see what they all are) and then we'll briefly discuss each one.

- Where are you going to put it? Is there enough space/volume? Is it convenient to the expected users of the system(s)?
- Is the floor of your space strong enough to support the weight of all that iron (no kidding!)?
- How are you going to provide power to all the nodes and switches and so forth[2]?
- How are you going to *remove* all that power when is released in the form of heat during operation? Is there enough air conditioning?
- Are there other infrastructure requirements? Do you need to run additional network lines into the space? A phone line? A thermal kill switch?
- Do you need any additional security measures for the space?
- Don't forget about *physical* network support, as in cable trays, places to run wire neatly, connections to any requisite LAN or WAN.
- What about a handy place to work on nodes? They won't run forever; you'll be in there fixing them.
- How are you going to *pay* for the recurring costs of running the nodes, and the amortized costs for renovating or "renting" the space it requires so that it has enough power, cooling, strong floors, security, network trays, workbench and tools and so forth?

[1] And I don't mean what beowulfish name you're going to give your beowulf when you're done, although that is certainly important.

[2] Before you even start to think about this in detail, I'd strongly recommend a trip to http://www.faqs.org/faqs/electrical-wiring/part1/ to learn a little bit about electricity and how it is distributed. I cover some of it in summary below, but you should take this subject fairly seriously.

"Hmmm," you say. "Spent so long thinking about the *programs* and the *node hardware* that I forgot about the physical requirements of all those nodes." I thought so. Let's take them in order.

11.1.1 Location, location, location

What constitutes a suitable location for a beowulf varies wildly, as one might expect given that beowulf designs vary wildly. You might be getting only eight nodes in mid-size towers, or you might be getting 128 nodes in racks. You could be getting eight nodes in a rack or 128 nodes in mid-size towers. You could be getting lots of nodes in both racks and towers. You could even be building a "blade" style beowulf with 32 nodes inside a single *chassis* mounted in a rack (times as many chassis as you like).

God knows, you could even build a beowulf by buying motherboards with CPU and memory, adding NICs, and hand-mounting them in e.g. an ordinary filing cabinet (fitted with a power supply, some cooling fans, and some spacers). With rack (and case) prices being what they are, I've come dangerously close to building a beowulf exactly this way at home and (as PXE/bootp NICs become ever more commonplace) I may yet do it. It would actually be kind of fun...

To some extent, what you want to get *depends* on the available space. If you only have a broom closet to fit the cluster in, your choices for a large number of nodes are highly limited to racks (presuming your closet has LOTS of air conditioning and electricity) or maybe rack cases full of relatively low-heat-producing single board computers. If your cluster is going to sit in a lovely gymnasium-sized raised-floor cluster facility with hot electricity and cold air delivered anywhere you like, you've probably got the money for racks but can obviously use anything you want.

In addition, you might be building the beowulfish cluster just for yourself, and want it in or near your primary workspace. Or you might be building it for a group of users and need it accessible to the group. Or it might even be a "public" facility and need to be accessible to several groups.

Even noise can be a factor. Our cluster room, between the 3-meter-cubed heat exchanger and blower in one end and several hundred CPUs (each with their CPU fan and an associated power supply and case fan all running at the same time) has roughly the noise level of a 747 taking off – on the outside of the plane. Not *quite* painful, but it's trying. One can hear it through a solid wood door and concrete block walls fifty feet or more down the hall.

Your computer cluster will need a space that meets your own particular constraints of accessibility, noise and security isolation, node design and denstity. Let's start with the simplest issues; how much space will it require?

Room for your Beowulf

Obviously, you need enough room. If you're only talking about four to eight nodes, you might well be able to build it in your office (presuming that your office is no smaller than mine). In fact, my current home beowulf is "in my office" – four nodes under desks in various rooms of the house, plus my own desk node and a small stack of "dedicated" nodes shelved next to the desk. I could easily stack up six to eight mid towers next to my desk, and have enough power and AC to service them (they live in recently renovated space that I engineered for a small cluster). I probably couldn't stack up sixteen, though, for a variety of reasons.

If you are planning to stack more than two units high, you will likely want shelves so that you don't have to unbuild your whole stack to get to a single node in the middle for service. My mini-mid towers are roughly 8x18x18 (inches), so I need at least two feet of space from a wall (to allow for cords and stuff) 16 inches wide and around 36 high to build a four node stack. With open shelves, I'd probably add two inches horizontally and four to six inches vertically to make it easy to get nodes in and out and allow a bit of air to circulate around the boxes.

A BIG shelved beowulf is most conveniently built well away from any wall, so that you can walk right around it. That way you can easily get to the back and do the cabling, can slide units in or out for service, and can keep air flowing through the whole thing to keep it cool. The same is generally true for rackmount beowulfs – even though the racks are often on wheels or casters, it's preferrable not to have to move them to get to either the front or the back.

Make *sure* that the floor of your space is in fact strong enough to support your beowulf. A loaded PC case might weigh anything from 10 pounds to 40 pounds (power supplies, especially, can be very heavy) and the shelving (if any) adds to the weight. Stack up enough of them on a few square feet of an upstairs room of a woodframe house and you can make the house quite unhappy. Remember, it's so embarrassing[3] when all that hardware (and you) plunge down on your pets and children in the living room in the event that you fundamentally overload the structure – I personally just hate it when that happens.

Even in an office-type building, if you crank three fully loaded full-size racks into a small office you can have a similar experience. Estimates published on the beowulf list for the weight of a fully loaded rack (I asked!) can range anywhere from 700 to perhaps 1500 pounds and even if nobody gets hurt when it crashes through the floor it would be an *expensive* mistake. In a lot of cases, organizations with a rackmount beowulf will also have a big UPS, which can also weigh in close to a ton in a footprint similar to that of a rack (1600 pounds on about 3 feet by 3 feet was reported on the list).

Another thing to remember is that you almost certainly want it to be twice the size that you think you "need" it to be at first. You need room for the physical nodes. You need room to walk *around* the physical nodes, attaching wires, punching switches, and generally looking busy if the boss comes buy. Assuming that you're not the boss, of course. Room for a desk or table capable of holding a monitor, an office chair, even a small workbench with a trusty electric screwdriver, multimeter, portable flashlight, and some bins for screws and whatnot is called for (and you should start scrounging for all of this stuff if

[3] You could just die from the humiliation.

you don't already have it handy). You need room to assemble and disassemble nodes, room to sit and work on node software via a workstation sitting on the desk, room to store various things. Finally, the robustness of your beowulf to failures in cooling depend mightily on how big the room is – how much "thermal ballast" does it provide. We'll talk about this further below.

Once you've gotten the *size* and the *strength* of the space worked out, you are by no means done. The next thing to work out is how to *feed* the beowulf what it needs to live and how to *remove* its waste products[4].

11.2 Power and Cooling for your Beowulf

Once you have located a space that is big enough and convenient to the administrators and/or users[5] and that can hold all the systems you plan to put there without overstressing the physical structure, it is time to think about **electrical power** and **air conditioning**. You *must* think about the two together, because the amount of one you require determines the amount of the other you require, and you need a lot more of both than you think you do.

A rule of thumb to use in estimating your power requirements is to assume perhaps 150 Watts per (Intel or AMD) node. This is *only* a rule of thumb - if you get dual CPU nodes with all the memory they can hold and a big power supply and add a big, fast disk and a CD ROM and four network cards and a video card and an extra fan, you might need twice that. Certain alpha nodes tend very power hungry. Basically, heat tends to rise at least roughly proportional to clock, so as CPU clocks continue to increase, so will heat production. On the other hand a "stripped" diskless node, running a relatively low-clock CPU, might end up only drawing 50 or 60 Watts.

However, you are in for a big surprise. The numbers above are for RMS average power consumption. If PC's had a "power factor" close to unity, as do (for example) electric incandescent lights, one could easily take those numbers and figure out how many nodes can go onto a single 20 Amp, 120 Volt circuit.

It also wouldn't matter much how the multiple circuits that feed a power pole in a cluster room were wired. One could wire the poles with three phases of a three phase Wye supply (described in more detail below) that shared a common neutral. Everything would be safe, and easy, and would work reliably for a long time.

Alas, modern switching power supplies have a power factor that isn't too close to 1. In addition, there are *details* of the way that they work that make a *huge difference* in the number of systems you can safely plug into a single circuit, the longevity of the nodes themslves, the safety of the supply power lines, and more. For that reason, it is time for a fairly detailed excursion into Your Friend, Electricity.

[4]No, beowulves are not alive. However, they come pretty close.

[5]You *do* care about the convenience of your users, don't you?

11.2.1 Your Friend, Electricity

Once again, if you are only planning on building a "small" beowulf (less than or equal to 16 nodes) you don't have to worry *too* much about power as most homes and businesses have circuits that can provide 1500-2000 Watts (15-20 Amps at 120 Volts) without blowing a fuse or breaker. Obviously you should check (thinking about other things, like a monitor and room lights, that might also be on the circuit) but you are likely OK.

With any more nodes than this, you are likely to need multiple circuits. You will also very likely need to have the room wired to obtain them, as (unless it is already a computer equipment room or a ex-machine shop or something) most rooms don't have multiple circuits already installed – they can actually be a bit dangerous in a home where somebody might mistakenly assume that because the lamp went off when they switched off a breaker it means that the next receptacle over is actually dead.

Anticipating that some of the folks who read this are expectant hobbyists or amateurs when it comes to electrical engineering, it seems like a good idea to learn a bit about electricity at this point. After all, electricity is one of those areas where what you don't know can kill you. Fairly easily, actually. Be Scared.

Let's discuss How Electricity Gets Around[6].

Electricity is typically delivered to your home or office as a 60 cycle per second (Hertz or Hz) alternating voltage from a step-down transformer (from a much higher voltage) outside the building. In general, it will come in as either two-phase (home) or three-phase (Y,office) with each phase at 120 (rms) AC volts above ground. By the time this 120 VAC gets to where it is going to be used, it often has dropped to only 110 VAC because of the resistance of the distribution wires, hence its generic name of 110 VAC.

To get a 110-120 VAC circuit, one connects a line with one of the phases through a fuse or circuit breaker to the black wire of a standard cable. The white wire is connected to a grounded stake or sometimes the plumbing. The bare copper (ground) wire is also connected to the grounded stake, but should *never* be used to deliberately carry current according to most electrical codes.

To get a 240 VAC circuit, one runs one 120 VAC phase on the red wire, and the opposite phase (of a two phase supply) on the black wire of appropriate cabling. Both are colored to indicate that either wire will provide 120 VAC with respect to white (current carrying ground) or copper (safe ground) or with respect to your delicate and easily damaged human body in contact with just about anything connected to the ground. So don't touch them if there is the faintest chance that they are "hot". Don't touch the white current carrying wire either – under certain circumstances it can carry enough voltage to kill you.

Kill you? Did I just say that? I did. Electricity is very dangerous and will kill you in a heartbeat[7]. Electricity can also start fires very easily, and fires

[6] In the United States, anyway. Sorry, I know, you probably live in Europe or India or Korea or South America, but I don't know anything about electricity and electrical codes there and hence couldn't help you anyway. If a volunteer from the beowulf list sends me details about other countries, I'll certainly break this section up into subsections, one per contribution.

[7] By stopping your heart; not, as a general rule, by cooking your brain like a hot dog unless

can also kill you dead. The best way to get your beowulf's space wired is by a certified professional who knows your local codes and is a lot less likely to come up with something that produces a blast of sparks when the breaker is thrown.

If you have three phase (Y or Wye, which is fairly commonly provided to businesses or industries but not common in homes) electricity, you can get a "sort of" 240 volt circuit out of it by running between any two of the three phases. The phase difference is only 120° instead of 180° so one ends up with only 208 VAC or so between the wires. This is enough to run most 240 VAC devices simply because the manufacturers aren't fools and know that Y/Wye supplies are fairly common. This is also true for a lot of computer equipment that requires 240 VAC (like some racks or uninterruptible power supplies (UPS) or some big-iron computers).

The thickness of the wires used to distribute the electricity and the length of the run from the primary distribution panel dictate how much current you can safely pull through a circuit. As a general rule (according to most local codes), 14 (for up to 15 amps) or 12 gauge wire (for up to 20 amps) is used in household dwellings to move electricity up to 100 feet. 10 gauge carries up to 30 amps (for e.g. air conditioners or the like). 8 gauge up to 40 amps. The smaller the gauge, the thicker the wire, the more it can carry without getting too hot. To go farther than 100 feet, one typically goes up a size (or more) of wire.

From this you can see that if you have a "large" beowulf, you will almost certainly need multiple circuits in the room (typically 20 amps each) and in many cases these circuits will have different phases. This means that if you are foolish enough to connect a black wire from one circuit to the black wire of another circuit, you could be basically shorting out 208-240 VAC. Amazingly enough, this happens (sometimes inside racks or computers that have more than one plug that manufacturers somehow assumed would always be plugged into the same circuit) with predictably spectacular results. This is just one of many reasons to have reliable fuses or circuit breakers in each and every line.

Once the electricity has made it to the room, there is no real difference between installing a bunch of receptacles in the wall for each circuit or just one or two and plugging power strips (with appropriately heavy gauge supply wires) into them, and the latter is likely more scalable and convenient. Just don't overload the circuits themselves and avoid thin extension cords and the like. Electricity "likes" to run over nice, fat wires and really hates it when it's squeezed down into a thin, scrawny wire. It responds by making those thin wires hot, which wastes energy, drops the voltage at the appliance, and can be dangerous.

You may want to think about uninterruptible power supplies (UPS) and power conditioning. In my area, the power goes off fairly frequently for tiny little times like ten seconds. This is just enough to cause all of your kitchen clocks and coffee makers to reset, and is plenty long enough to hard-crash your computer(s) as well, which is most annoying if you've been running a calculation

---

you are messing with really high voltage and current. 60 Hz turns out to be a *bad* frequency because it interferes with the biological frequencies that keep your heart cranking along. Oops. 1000 Hz would have been a much safer choice.

for a day or two (or longer!) and have to start over. Almost any kind of UPS can keep a computer up through these short outages.

More expensive UPS can provide a degree of power conditioning and surge protection, which is also useful when you have many nodes and want maximal hardware reliability. Some of them also have other clever or desirable features, like the ability to control them and cycle the power remotely via a serial port connection or the like. This can sometimes save one a trip into the cluster in the middle of the night or can allow you to reboot while on your ski vacation in Europe, if that sort of thing is worth it to you (the bells and whistles aren't cheap).

So fine, you've got your space, it has room, the floor will hold all your systems (and you and your desk and your stereo), you've got electricians running one 20 Amp circuit in for each 16 nodes (or thereabouts). There's just one last major problem to worry about. You're delivering a lot of power to the room to run all those machines. When they're done with all that energy, they give it up as heat. Every watt that goes *in* to your computer room has to come *out* in a steady state.

Believe me here, I'm a physicist. Think of your 16 node beowulf as a 1600 watt space heater or 16 100 Watt light bulbs, and you won't go far wrong. 1600 watts is the *rate* at which energy is being delivered into the room[8]. If you don't remove all that energy at the same rate, it will build up. As it builds up, the room will get hotter and hotter until the temperature difference between the inside of the room and the outside of the room is big enough to drive all the heat out through the walls.

This may or may not happen before all your computers melt or catch on fire and turn into an expensive little puddle of metal and epoxy. Or just break, which is actually more likely but not as impressive. The former *can happen*, though – as you may discover the hard way if you are foolish enough to put 128 nodes (or approximately 13,000 watts) into a small, closed room with no kind of thermal kill switch and the air conditioning fails.

Once again, most rooms in most houses or office buildings can probably handle as many as eight nodes with their existing air conditioning arrangements. In my house, for example, my office gets a bit warm during the summer with five nodes (two with monitors) for around 700 watts, plus a couple of lights (150 watts more) plus a couple of warm bodies (200 watts more). 1000 watts in a 10 foot square room with a door and the house air conditioning set in the low seventies keeps the office temperature in the high seventies, but I can live with that and so can my nodes.

Sixteen nodes, of course, would be intolerable unless I added a window air conditioning unit (or unless I spread them out throughout the house). Once again, you'll have to work this out for however many nodes you plan to have, but if you have more than a very few nodes you *must* work it out.

A useful True Fact is that air conditioning is usually bought in "tons", but

[8]For the physics challenged, a watt is a joule per second. A joule is a unit of energy, like a BTU or a calorie.

any sane measurement of power being delivered to a room will be in watts (or maybe kilowatts). So, MaryLou, what's Ton? A ton of air conditioning removes enough heat to melt a ton of ice at the melting point (0° C) in 24 hours. To calculate the power this represents is a pain in the butt, however straightforward[9] and the result is that one tone of air conditioning can remove almost exactly 3500 watts continuously from a room.

So, in an ideal universe we could run perhaps 32 nodes per ton of available air conditioning (to stay a bit on the safe side). A 128 node beowulf might need four tons of air conditioning (depending on the actual power required by the nodes, which may well vary). However, reality might well be less than ideal – if your machine room is considerably *cooler* than its ambient surroundings, or has a large sunny window, or has a lot of electric lights, you may not be far enough on the safe side. Heat can flow *in* to the room from any of these sources and 1 square meter of sunny window can let a *lot* of heat into a room on a hot and sunny day.

I'm tempted to expound on the additional *power* needed by all that air conditioning, but that depends on the efficiency of your air conditioning unit and the temperature of the outside air and all that. A reasonable estimate is that you'll have to buy a watt of air conditioning power for every three to five watts of power consumed in your beowulf. Ahhh, physics. A wonderful thing.

Let me remind you one last time that if there is any chance at all that your air conditioning can shut down while your computers are still operating and they are not in a *large* room with plenty of circulation, you should think seriously about some sort of thermal kill switch. Computer hardware breaks or even catches on fire if it gets hot enough, and I can tell you from bitter experience that the temperature in a smallish closed room (in an otherwise cool building) will go up to well over 100° Farenheit in a remarkably short time if there is more than a kilowatt being released inside with no ventilation or air conditioning. The temperature inside the cases will be considerably higher, and the temperature of the CPU and memory chips and hard drives higher still.

We're now done with the serious stuff. I'll wrap up this section by reminding you to think about other kinds of infrastructure that you might want to provide for your beowulf room if it is in some sort of organization; fiber or copper lines to your organization LAN switch or router, for example, or connections to printing facilities. A phone (or two) is often nice, possibly equipped with a modem and terminal or network server if you plan on managing remotely (as in from someplace network-inaccessible).

Finally, you may want to think about physically securing the location. You've just built a pile of PC's that (however cheap the nodes) is worth thousands, possibly hundreds of thousands of dollars. It would be a shame if you

[9] Aha! You thought that I'd present the calculation here, didn't you. Admit it. You're just being lazy, and I'm tempted to tell you to get out an envelope, but you probably don't remember the latent heat of fusion for water (333.5 kJ/kg) or the number of kilograms in a pound (0.4535) or the number of seconds in a day (86400) and all that. So one ton of air conditioning can remove $2000 \times 0.4535 \times 333.5/86,400 = 3.501$ kilowatts from a room continuously

came in after a weekend to discover that an entrepreneur with a pickup truck had disassembled and made off with a large chunk of them.

I wish that I could say that this is very unlikely, but we've had computers stolen (including one high end beowulf node) from just outside our beowulf room, which is itself located on a low-traffic hall inside a generally locked building with a carded lot. We're likely going to move our beowulf room to new digs on a NO traffic corridor that you have to have a building map to find. So think about locks, traffic patterns, access both day and night, and don't make it too easy for an "entrepreneur" to make off with your hard-earned nodes and support hardware.

The answer, fortunately, is that it is not difficult at all to build, and once built and configured, it is extremely easy (and cheap!) to maintain. Linux (or at least some sort of Unix) expertise is obviously very useful, but most linux distributions fully support generic cluster computing "out of the box". The most difficult single things to master are how to implement a *scalable* installation mechanism for your cluster (or LAN), and how to largely automate software maintenance for your cluster (or LAN) so that you do work once, and it is automatically applied to all the nodes (or workstations) you manage.

Why do I keep putting down nodes (or workstations)? Hmmm, good question! I suppose the answer is that from *one* point of view a generic compute cluster can be thought of as a *LAN consisting of specialized workstations*. In particular, workstations with no X or GUI installed, that indeed might not even have video and a keyboard installed at all, that are missing sound and games and office tools and a whole lot of user applications, but that *do* have compilers and other development tools, a wide range of application and development libraries, specialized libraries and toolsets for supporting e.g. PVM or MPI computations, and perhaps some specialized node monitoring daemons or batch job management tools installed.

Nearly all of this could equally be installed on a workstation, and if you run cluster nodes in your workstation LAN, you are *very likely* (and wise) to go ahead and install all the cluster tools but perhaps the batch schedulers on your workstations as well, so that the *only* difference between a workstation and a cluster node is that most "desktop user" interactive/user interface components are missing on the latter.

Note that this is *not* the strategy adopted by the "true beowulf" package builders[10], who install custom kernels and tools to make cluster nodes look like "CPUS" in a big multiprocessor system with a unified PID space and transparent job distribution and management. In this latter approach, nodes are *not* workstations, and you can't "log into a node" any more than one can "log into a CPU" on a MP system.

[10] Scyld, at http://www.scyld.com, is commercial, Clustermatic, at http://www.clustermatic.org, is non-commercial. Both are open source; both are based on Erik Hendriks "bproc" program. The Scyld solution comes with hot and cold running commercial support, but costs money. The Clustermatic solution comes with the usual hot and cold running support-by-other-users, and is "free". Ya pays your money (or not) and takes your choice...

This suggests that it is time for a pretty fundamental split in the discussion. All those who want to build a beowulfish cluster on top of their existing LAN, integrated with and possibly even transparently including their desktop workstations, creating nodes that are basically specialized, particularly *simple* workstations (that one can log into to run jobs or do whatever you like, just as one could a workstation) please move one full pace to the left. Unless, of course, you happen to be sitting down, or moving to the left would cause you to fall off of a tall building and die, can't have that.

All the rest of you, who want none of this "workstation cluster" crap and want to build a *beowulf*, pure and simple, similarly step to the right, if only metaphorically. Wishy washy ones can stay where they are and read both of the following sections to figure out which one they might be, or might become, and how.

11.3 Building "Workstation"-like Nodes

Using the most advanced installation techniques available (and there are a number of distinct approaches one can take in all the different distributions) it is possible to definitely automate node installation in Red Hat and Debian and probably all the rest as well but I haven't tried them or gotten explicit instructions from somebody who has. In addition, there are at least two distinct "beowulf in a box" open source projects out there as of the instant I'm writing this (one more "open" than the other, but both valuable).

This chapter is organized into three sections. The first one covers what you need to do after you've more or less determined your beowulf architecture and scale but before you actually purchase the components. The second one covers the assembly of the components, and includes a few clever tricks and ideas that have been contributed over the years on the linux list, as well as a few gotchas. The third section covers how to take care of the beowulf once it is built and running. It is pretty boring, as once it is properly installed linux is boringly stable; most of the required maintenance is just standard Unix maintenance of the server(s) (backing them up and so forth), fixing hardware if and as it breaks, and possibly helping and educating users.

11.4 Building the Beowulf

This part is pretty easy, once you've prepared a home for it. Of course, this is what Sterling, Salmon, Becker and Savarese wrote the whole book *How to Build a Beowulf* to tell you to do, so maybe it isn't *that* easy. I'm going to give you the relatively short version here and refer you to the SSBS book (and or a linux book) if you want still more detail.

Basically, you start by setting up all your nodes physically. This may be stacking them up on the floor or on shelves or on a table. It might be assembling a rack. It might be just uncrating a turnkey beowulf if you shop at Paralogic (www.plogic.com) or Alta Tech (www.altatech.com) or any of the other turnkey beowulf makers.

Plug them in, cable them up (which means connect all the NICs to their switch, generally, and you're ready to install. At this point what you do depends very much on how you configured the nodes, which in turn probably depends on what you planned to do at this point.

Uhhh, say that again? Well, hopefully you READ this whole book before ordering all your nodes, didn't you? So when you made certain configuration decisions you did so knowing what you planned to do when you got here. So all I have to do now is tell you what those decisions might have been and what to do if they were, or something like that. I'm confused myself by this point.

Let's do this by going from the easiest but most expensive and perhaps most time consuming to the cheapest but most difficult ways of managing an installation.

The easiest, of course, is to buy a turnkey beowulf from one of the aforementioned vendors (or one from the lists given in the appendices; I'm not on the take of Alta Tech, although in the spirit of full disclosure I have to say that Doug Eadline of Paralogic did indeed give my kids Extreme Machines tee shirts last year at Linux Expo). If you did this, I rather imagine you plug it in and turn it on and go about setting up accounts and all that. In any event, if you did this you don't need my help as you likely bought help (support) along with your beowulf.

11.4.1 Expensive but Simple

If you are a real neophyte in all senses of the word (a linux neophyte, a parallel computing neophyte, a beowulf neophyte, a network manager neophyte) and need to pretty much learn *everything* as you go along, you're going to want to either stick to the following recipe. If you did your work very carefully and know that you really need to build one of the cheaper designs discussed next, you should still build yourself a very small beowulf (perhaps four nodes) out of either systems and parts at hand or systems you plan to recycle into stripped nodes or server nodes according to this plan just to learn what you're doing.

That is, the following design isn't really very cost-beneficial as you buy some things your nodes don't really need and do a lot of work by hand, but it is still reasonable for small beowulfs or while you're learning. From what you learn you can understand how to implement the next design that scales a lot better in all ways.

This design has you install and configure each node basically by hand using the standard installation tools that come with whatever linux distribution you selected to use. You then put the nodes onto the chosen network(s) and configure them for parallel operation.

The other place where this design works as a prototype is if you are setting up a beowulf-style *cluster* that isn't really a true beowulf. In that case you'd actually configure each node to be a fully functioning standalone workstation, setting up X and sound and all that, and providing each node with a monitor and keyboard and room on a table or desk. You'd still install most of the "beowulf" software – PVM, MPI, MOSIX and so forth, and you'd still configure it for parallel operation.

In this "hand crafted beowulf" design, your nodes have to be configured to install independently. These days, that means that they probably need the following hardware:

- A floppy drive.
- A cheap, small (4 GB is small these days) IDE hard drive.
- A CD-Rom drive
- A generic SVGA card (I usually get $30 S3-Virge cards)

plus of course your NIC(s). Each node is then attached to your choice of the following:

- A KVM (Keyboard, Video, Mouse) switch, which in turn is connected to a single keyboard, monitor and mouse. KVM switches are available that are cheap (but fuzz a high resolution monitor a bit and don't work for PS/2 mice) or expensive (but keep the monitor clear and can manage all kinds of mice). The latter can be purchased to support all the way up to some 64 nodes, although they might add almost as much to the marginal cost of your nodes as a monitor, keyboard and mouse for each.
- A monitor, keyboard and mouse for each. That is, you're building a NOW (network of workstations) or COW (cluster of workstations) as opposed to building a "true beowulf". Big deal. It will still work like a beowulf for anything but moderately fine grained synchronous parallel code and you can use the workstations for all sorts of useful (but not particularly CPU or network intensive) things while it is doing parallel computations.
- A moderately portable monitor, keyboard and mouse, perhaps on a cart. You plug this into the nodes only one at a time of course, installing one, then the next one, then the next and so on.
- One of several moderately expensive specialty cards that let you use (e.g.) a serial console for the original install. Expect to pay three or four times the cost of a cheap SVGA card.

The installation procedure is then very simple. You plug your distribution CD into the CD-Rom drive, the boot floppy into the floppy drive, (if necessary attach the portable monitor and keyboard to the appropriate ports) and boot. You will generally find yourself in your distribution's standard install program.

From there, install a more or less standard linux according to the distribution instructions. You probably have more than enough hard disk space to install everything as it is hard to buy a disk nowadays with less than 4 gigabytes (which is way plenty) so don't waste too much time picking and choosing – if it looks like it *might* be useful install it, or just install "everything" if that is an option. Be moderately careful to install all the nodes the same way as you really want them to be as "identical" as possible.

Be sure to include general programming support (compilers, libraries, editors, debuggers, and documentation). Be sure to include the full kernel, including sources and documentation (a lot of distributions won't install the kernel source unless you ask it to). Be sure to install all networking support, including things like NFS server packages. Sure, a lot of these things will never be needed on a node (at least if you do things correctly overall), but if they *are* ever needed it will be a total pain in the rear to put them on later and space is cheap (your time later is expensive).

Be sure to install enough swap space to handle the node's memory if you can possibly spare the disk. A rule of thumb to follow might be to install 1-2x main memory. Again, if you are sensible (and read the chapter on the utter evil of swapping) you will avoid running the nodes so that they swap. However, in the real world memory leaks (MPI is legendary for leaking in real live beowulfs!), Joe runs his job at the same time as Mary without telling her, a forking daemon goes forking nuts and spawns a few thousand instances of itself, netscape goes berserk on a NOW workstation, and you'd just LOVE to have a tiny bit of slack to try to kill off the offending processes without wasting Mary's two week run. A system without swap that runs out of memory generally dies a ghastly death soon thereafter. It's one of the few ways to crash even linux. Be warned.

Finally, install your beowulf specific software off of a homemade CD or the net (when the network is up) or perhaps the CD that came with this book (if a CD came with this book). If you installed a distribution that uses RPM's (like Red Hat, SuSE, Caldera) this should be straightforward. Debian users will firebomb my house if I don't extend this to Debian packages as well, so I will. At this point in my life, I'd tend to avoid Slackware although we were very happy together for years. Good packaging systems scale well to lots of nodes, and scalability is key to manageability.

With all the software installed, it is time to do the system configuration. Here I cannot possibly walk you through a full course in linux systems management, and most of what you do is common to all linux or unix systems, things like installing a root password (you probably did this during the install, actually, and hopefully picked the same password for all nodes), setting up the network, setting up security and network services, setting up NFS mounts, and so forth. To learn how to do all this, you can use the documentation that came with your distribution or head on down to Barnes and Noble (or over to amazon.com) and get a few books on the subject. Be warned that the "administration tools" that come with most linux distributions suck wildly in so many ways[11] so even if you

[11] Let's see, they are slow, they are broken and don't work, they are clumsy, they *require* that you have graphical interface (the X console) working, they are broken and don't work, they are security risks, they prevent you from ever learning proper scalable systems management techniques for a beowulf, they are broken and they don't work right where you damn well need them to, they are utterly different on different distributions and hence what you learn isn't portable at all... Did I mention that they are often broken and don't always work?

use them to get started you *need* to learn how to do things by hand.

There are a few things you need to do a bit differently than the out-of-the-box configuration, and I'll focus on just these.

- Be sure that the latest version of the openssh package is installed on all the nodes[12]. Keep this revision up to date as aggressively as you can manage, as there are occasional security holes found in ssh and you want to be sure you are working with the latest patched release. The latest releases of ssh are also *much* easier to debug when something goes wrong with your setup.

- When you set up networking on a "true beowulf" node (one that is isolated from the main network of your organization by some sort of gateway node), use an IP number for a private internal network. Private internal networks are described in an RFC (if you know what that is or care). They are also described in the HOWTO on IP-Masquerading. I personally like the 192.168.x.x addresses, but you can also use the 10.x.x.x addresses (if you want to be lavish) or the 176.[16-31].x.x, which I can never remember. Remember not to assign the 0 address or the 255 address to nodes – that is, use only something like 192.168.1.[1-254] as a range. 0 and 255 are "special" addresses and can break things if used.

- Set up a common /etc/hosts or some sort of nameservice. There are good things and bad things about using NIS to manage system databases like this. It is likely that the bad outweighs the good – NIS can significantly increase the overhead of certain kinds of network traffic and network traffic is the *last* thing that you want to slow down in a beowulf. On a "true beowulf" most people tend to use a tool like rsync or an scp script to distribute identical copies of /etc/passwd, /etc/group, /etc/hosts, and so forth. However, in a NOW-type cluster with lots of users (and not particularly fine grained parallel code) NIS is a reasonable enough solution.

When you are done and have rebooted the node, it should come up accessible (via ssh) over the network. Once you can login as root over the net (*test* this) you can move or switch the monitor and keyboard to the next node.

With all of this established, and with ssh set up to permit root access to the nodes from the head node without a password, it is time to distribute common copies of things like /etc/hosts, /etc/hosts.[allow,deny], /etc/passwd, and your preferred /root home directory (I tend to like to customize mine in various ways and miss the customizations when they aren't there).

[12]Many beowulfers would disagree that ssh is necessary on the nodes, as they are typically on a private network behind a system that functions as a de facto firewall (all of which is true). Tough, they're wrong and I'm right. They can write their own book. Even on strictly functional grounds, rsh sucks (in addition to being, as Mr. Protocol poetically put it in Sun Expert 11.3, a "rampaging security hole masquerading as a convenient remote command execution facility"). It needs to die, die, die and it won't as long as there are still pitiful fools who still use it.

To do this, one can use something like rsync (with the underlying shell set to ssh, of course) or just an scp. Either way, you will find it very useful to have a set of scripts on the head node that permit commands to be executed on all nodes (one at a time, in order) or files copied to all nodes (one after another, in order). Some simple scripts for doing this sort of thing are in the Software appendix (and available on the web, since I doubt that you want to type them in).

I'd strongly recommend that you arrange for all nodes to do all their logging on your head node to make it as easy as possible to monitor the nodes and to make it as easy as possible to reinstall or upgrade the nodes. If all they contain is a distribution plus some simple post-install configuration files, you don't need to back them up as reinstalling them according to your recipe will generally be faster. This is a good reason to set things up so that the nodes provide at most scratch space on disk for running calculations with the full understanding that this space is volatile and will go away if a node dies.

When you are finished with this general configuration, one should have a head node (mywulf outside and bhead inside) that is also an NFS server exporting home directory space and project space to all the nodes. You should have a common password file (and possibly /etc/shadow file) on all the nodes containing all your expected users. You should have ssh set up so all your users (and root) can transparently execute ssh commands on all nodes from the head node or each other (root might only work from the head node). That is, "ssh b12 ls /" should show you the contents of the root directory without a password. You should have PVM and MPI (and possibly other things like MOSIX or a queuing system) installed on all nodes (probably via an NFS mount – there is little reason to maintain N copies of the binary installation, although with RPM or a decent package manager there isn't too much reason not to).

PVM or MPI should be configured so that they are can utilize all the nodes. How to do this is beyond the scope of this book – there are lots of nice references on both of them and one can usually succeed even if one only follows the instructions provided with both of them. With PVM, for example, you'll have to tell it to use ssh instead of rsh and decide whether you want to run pvmd as root (with a preconfigured virtual machine) or let users build their own virtual machine for any given calculation, which in turn may depend on who your users are and what sort of usage policy you have. Similar decisions are required for MPI. It is a very good idea to run a few of the test examples that are provided with PVM and MPI to verify that your beowulf is functioning.

From this point on, you can declare your beowulf open for business. Your work is probably not done, as I've only described a very minimalist beginning, but from this beginning you can learn and add bells and whistles as you need them.

This approach, as we've seen, more or less builds your beowulf nodes by hand. This teaches you the most about how to build them and configure them, but it doesn't scale too well. It might take you as long as half a day to install a new node using the approach above, even after you you have mastered it (the first few nodes might take you days or weeks to get "just right"). There has to be a better way.

Of course there is. There are several, and I'll proceed to cover at least two. The next example will be a bit Red Hat-centric in my description. This is *not* to endorse Red Hat over any other linux but simply because I'm most familiar with Red Hat and too lazy to experiment with alternatives (at least to the point of becoming moderately "expert"). It is certain that a very similar solution is possible with other distributions, if you take the time to figure out how to make it work.

11.4.2 Cheap, Scalable, and Robust

The *best*[13] way to build your beowulf nodes (as opposed to the easiest) will be to *learn* some things that enable you to improve the scalability of your installation. What exactly does that mean?

If you followed my advice (assuming you are new to all this) and built a small beowulf by hand-installing and configuring a few nodes, you probably noticed that you did a lot of things over and over again. You configure the disk. You pick the packages. You install the root password. You (possibly) set up the video, mouse, serial port, printer(s) and all that. You configure the network. You come along afterwards and copy the same set of /etc files from your master copy of them on your server. You also unpack the same set of beowulfish add-ons, and configure them the same way.

With all that repetitive work (and a computer handy, for God's sake!) one expects there to be a better way and of course there is. What one needs is a way of *encapsulating* all of those actions that were the same over and over again into a *script* (possibly with some variables) to do it for you.

If you are a Unix or linux expert, you're shaking your head and saying, "Well Duh...". On the other hand, if you are coming out of the Windoze or Macintoad world, you probably think that a script is the thing used to put on a play or a movie[14].

So it is, my friends, so it is, but the *particular* play we wish to put on is one where the computer itself is the actor while it "installs itself" according to our specifications!

Fortunately, linux comes with marvelously powerful scripting languages (like /bin/[ba]sh[15], perl and python[16], and still others that are arcane or archaic, depending on one's religious convictions and history.

[13] Oooo, fighting words on any linux list. So let me be specific to try to avoid stimulating a supernova in my neighborhood. This is an introductory book, right? So I mean the best way for relatively unskilled people to achieve a soundly designed, clearly scalable, simply manageable, and robust beowulfish cluster. If you know enough to argue with the word "best", relax. It isn't intended for you anyway. Best is relative.

[14] Although if I called it a "batch job", those of you who started with DOS 1.0 or 1.1 – like me, actually – would immediately recognize it.

[15] Stop smirking. One can do "anything" with a Bourne shell script, at least if one is willing to work hard enough. Well, almost.

[16] Stop fighting, you two! Put down that toolkit! I mean it!

Unfortunately, all the different linux distributions are, well, different. And the one place they are most annoyingly different is in how they lay out certain key files that one most certainly will need to configure, and in how they actually install themselves onto a system in the first place.

These differences are not insurmountable, but they make a truly proper and portable scriptset for installing a beowulf node a childish fantasy. Your odds are better of winning the Publisher's Clearing House Sweepstakes, or of being brained by a small meteorite as you step out your front door[17] than they are of writing something that works on any *three* of the available linux distributions. Alas, beowulf install scripts for distributions exist, but they tend to be crafted as one-of-a-kinds and therefore also tend to break even for that distribution when, for example, it goes up a major revision number.

Fortunately (we just did unfortunately, right, so we're back to fortunately) it isn't *that* hard to come up with a decent install paradigm that works with the strengths and overcomes the weaknesses of any given distribution, once you know what you are doing and now you do. To show you roughly how to proceed, I'm going to outline what I do that exploits the nice scripting features of Red Hat's kickstart install. Near-equivalent approaches can be constructed for most of the other distributions as well, where you have to do more or less of the configuration work yourself with external scripts.

On major advantage of the Red Hat kickstart approach is that one does *not* need (or want) a CD-ROM drive in a node in this approach – a floppy and hard disk suffice. One *may* not need a graphics adapter, although even a host bios that permits one to boot without one usually has to be booted with one first to set the bios to boot without one[18].

Now the point of kickstart is that Red Hat has encapsulated a large portion of the configuration tree that you followed setting up nodes by hand into a single file, where it effectively defines a "script" to be used by Red Hat's Install program. Using kickstart, one builds this configuration file that tells Red Hat's installation scripts how to build and configure a node, sets up an NFS exported directory containing all required RPM's, the kickstart configuration script, and builds any scripts to run after the RPM phase of the install to (for example) distribute /etc/passwd and the like.

A nice way to proceed (although not strictly necessary) is to set up the head node (which should already be built and configured separately by hand) to be a dhcp server. This server is configured to recognize the ethernet number of each node as it boots and deliver to that node its IP number and all network setup information and the instructions for getting to its kickstart file. Finally, one builds a Red Hat kickstart boot floppy, which is basically a standard boot

[17]Amazing Facts: Yes Virginia, humans have on multiple occasions been struck by meteors. On many of those occasions they have lived. Humans have (to the best of my knowledge) never succeeded in writing a single script that would install Red Hat, SuSE, Caldera, Debian, Slackware, TurboLinux (and the list goes on) in a beowulfish configuration. I rest my case.

[18]Meaning that one has to boot a host at least one time with an SVGA card and monitor and keyboard attached, which is a pain that adds thirty minutes to each node install. It's your time, but I'd just buy the damn $30 card if I were you.

floppy that times out into kickstart instead of an interactive install. Detailed instructions for setting all of this up (with code and examples) are given in the Software appendix and are available online at URL's given therein.

When all goes well, a node install from the point where it is plugged in and connected to the network (with *no* monitor or keyboard attached) looks something like:

1. Boot the node with the kickstart floppy.

2. On the head node, note the (rejected) dhcp request. Create an entry in the dhpcd configuration file for the host with the ethernet address observed in the dhcp log. Restart dhpcd.

3. Reboot the node with the kickstart floppy (a hard boot is fine). Wait a time long enough for the install to have definitely finished (typically 20-30 minutes). You can work on more nodes in parallel while waiting. With a bit of practice, you can install a node every five to fifteen minutes with three or four nodes always in the process of being installed at once.

4. Remove the kickstart floppy and hard boot the node.

If all has gone as it should, the node should come up fully installed, gifted with a unique identity, and ready to accept ssh logins over the network. If you wrapped enough of the post-install configuration up into scripts you may *never have to touch* the host administratively again except for routine maintenance.

The advantages of this strategy are manifold.

- You can reinstall a node at any time by rebooting from the kickstart floppy. If the node is not used to store data, you don't need a backup at all as it will be faster to reinstall completely than to mess with restoring a backup. Time to reinstall is typically about twenty minutes.

- You can replace a node transparently by rebooting a replacement from the kickstart floppy (either reusing the old NIC or making a new entry in the dhpcd configuration table). Time to reinstall is still far less than an hour, even if you have to move the card.

- All nodes are "identical" and have precisely defined sets of binaries, libraries and applications. This is very good for scaling.

- Upgrades are no more complicated than an install. If you can upgrade from RPM's, this can be accomplished via a nightly cron script (drop the RPM's into a special NFS mounted directory and they'll be on the nodes by morning). If you are doing a full distribution upgrade, you just reboot from the kickstart floppy once you've converted the node kickstart configuration file for the new distribution. Upgrading a 32 node (or more) beowulf can easily be accomplished in a day.

- You save at least the cost of a CD-ROM drive (and a bit of power) per node, the cost of maintaining node backups, and most of the human cost of installing or managing a node.

As you can see, kickstart or kickstart-like automated script-driven installation tools in general permit one to build "a node" as a virtual description that can be reused for all nodes. This scales excellently well and lets you work on solving problems *once* with the assurance that your solution will appear uniformly on all nodes in due course. In managing networks in general but beowulfs in particular, heterogeneity is evil. Try to ensure that all your nodes are boringly identical in hardware and are running a boringly identical software setup as well. Every single thing you have to do "by hand" or on the basis of a node's particular identity is work that will become a hideous burden over time and retard upgrades and so forth. Only your head node should be at all different.

Details of the kickstart approach

In order to use the approach above in the most automated fashion possible, one would like to make a kickstart floppy, that is, one that runs kickstart by default after a fairly short timeout. This is essential if you want to be able to (re)install without a monitor or keyboard attached.

It is simple to do this in Red Hat. Use the following steps:

1. Copy the bootnet.img from the ./images directory of Red Hat distribution you are using as your basic system to /tmp/bootnet.img
2. Mount this image via loopback. As root, a sequence like:

   ```
   mkdir /mnt/loop
   mount -o loop /tmp/bootnet.img /mnt/loop
   ```

   should do it.
3. Edit /mnt/loop/syslinux.cfg. Change the default from "linux" to "ks" and the timeout from 600 (or whatever it is) to as many tenths of a second as you want to have to override the default if/when you might need to – I like 50 (5 seconds) or 100 (10 seconds) but no more.
4. Umount /mnt/loop
5. Copy /tmp/bootnet.img (which should now be modified) onto as many floppies as you like via:

   ```
   dd if=/tmp/bootnet.img of=/dev/fd0 bs=1k
   ```

The floppies thus created SHOULD start up and display the usual initial panel message (which you can also alter if you like by editing /mnt/loop/boot.msg when the image is mounted but there is little point). After a timeout of 5-10 seconds, they should boot into a kickstart install. If you have dhcpd.conf records like:

```
host b01 {
        hardware ethernet 00:00:00:00:00:00;
        fixed-address 192.168.1.1;
        next-server install.my.domain;
        filename "/export/install/linux/rh-7.1/ks/beowulf";
        option domain-name "my.domain";
}
```

(where you have to fill in the ethernet address of EACH node in question, giving each one its own unique node name and fixed address, probably in a private internal network space as shown). The file pointed to is the kickstart file for the node and should contain things like a url or other path to the distribution and so forth. Some example kickstart files are given in an appendix.

There are still other games that can be played with kickstart and lilo working together. For example, lilo can be passed a parameter that instructs it to boot a running system directly into a kickstart reinstall. The last command of the kickstart reinstall can reset lilo again to boot the system normally. Once your systems are set up and running, you may never again need to boot from a floppy to do a reinstallation. As long as your nodes remain "clean" (are not used to store real data that requires preservation or backup) they can be reinstalled in any four or five minute interval of downtime to upgrade or repair with a simple network command a routine script.

There is yet another, still more sophisticated approach to building nodes. This one results in the cheapest possible nodes (which often means that you can afford more of them). It is, however, in many ways the most difficult to get working, and the resulting nodes are in some ways the least stable. For many problems and budgets, though, it is worth the effort to figure it out and make it all work.

What about other distributions?

Hmmm. A fairly obvious problem emerges. True, I can type like the wind itself, fingers dancing on the keys with a life of their own, my interior monologue spewing out with surprisingly little editorial revision to appear, as fully marked up prose on the screen. True, my experience in Unix, cluster computing, linux is great, my expertise is vast and almost boundless[19]. Amazing as it may be, and to my great sorrow, I'm not an expert in everything – yet – and given that (however fast) my typing in new stuff that I learn proceeds linearly with time, while the amount of new stuff I *need* to learn and eventually type in increases exponentially, I will almost certainly Never Catch Up. This is especially true given that you who are reading these words almost certainly are doing so *for free*, leaving me to manage mundane matters like feeding my children and paying for beer on my own. All God's chilluns gotta eat, and we gotta work at paying enterprises to buy food.

[19]Stop that! Stop that laughing! You're the one reading *my* book, after all...

Consequently, I am in perpetual need of help. Not just psychiatric help, either. Material help, in the form of material that can easily be included in this book *without* my having to learn all about it and type it up. The following little snippet is one of many such contributions, concerning tools that can be used in other distributions to automate beowulf installation along the lines of kickstart in Red Hat. To prove that I'm not a Red Hat bigot (or being paid off by their board of directors, however attractive that would be and bribeable I might be) please note: (from Thomas Lange ¡lange@informatik.Uni-Koeln.DE¿)

> The other and more important note is on how to build a beowulf. You only mention kickstart. Have a look at fai (http://www.informatik.uni-koeln.de/fai/). It's the fully automatic installation tool for Debian. There's also a chapter on how to build a beowulf using fai at http://www.informatik.uni-koeln.de/fai/fai-guide.html/ch-beowulf.html

I'm *trying* to keep track of the many references like this that have been mentioned on the beowulf list or in private communications like this, but given the busyness of my life (and the aforementioned fact that I basically make no money from this book:-) it may be a while before I ever catch up on them all. Like an infinite while...

In the meantime, fee free to fire stuff off to me that you think should be in here somewhere, or to point out (as Thomas also did) that nobody uses ssh 1.x any more and it is a security bugfarm so I should probably fix the text (written years ago) where I advocate its use, or the equivalent. This field is a moving target at best and this book perpetually obsolete here and there, and I need all the help I can get adding new relevant material or targeting the old obsolete material for stat revision.

11.4.3 Cheapest and Hardest: Diskless Nodes

Diskless nodes, you say? But, but, but, how do they boot? What do they run? How do they work?

I'm glad you asked.

Actually, in all parts of the Unixoid computing universe *but* that part occupied by the Wintel-Macintoad industrial conspiracy[20] diskless operation via NFS has been a standard feature in widespread operation for longer than I've been managing unixoid systems (some fifteen years, with some years of DOS before that). Sun Microsystems, in particular, for years sold workstations *designed* to operate as diskless computers. They didn't have a disk. They didn't want a disk. They didn't need a disk. And all of this was on a far older, far

[20] Actually, I'm joking, here, sort of, in case you were taking the word "'conspiracy" too seriously. Diskless operation requires a *sophisticated* operating system and *seamless* remote disk services, and, well, who would "conspire" to keep their operating systems hopelessly unsophisticated and incapable of real networking? It must be a cruel accident; nobody could be that stupid.

slower network than we enjoy today at a time that 16 megabytes was a *lot* of memory.

Nowadays, with 100 megabit per second switched networks the minimal beowulf standard (in most cases), 500+ bogomip servers, memory available at less than $1 per megabyte, diskless configuration works, it works *well*, and it saves you at least the cost of one hard disk per node, which is (these days) approximately $100 per node, which (these days) might buy you six nodes for the cost of five or better. Diskless nodes can make good economic sense.

The linux kernel is perfectly capable of diskless operation, and has been for several major revisions now. There are only one or two things that make diskless operation more difficult than it really should be, and it is these things that make this the *second* best way to run a beowulf for most people (presuming that most people reading this book are not, or at any rate are not yet, linux gurus).

One of these things is that (conspiracy or not) "personal computer" makers (as opposed to "Unix workstation" makers like Sun and SGI) generally don't install BIOS's that are capable of booting over a network device. This, in turn is at least partly because the network device in a PC generally is made by a third party and requires a driver that doesn't live in the bios because of the lack of a uniform network device API. This leaves one with a hideous chicken and egg sort of problem. To get a device driver for the network card into a system, one needs a disk. If one had it, one could boot diskless (and load a network device driver without a disk).

Oops.

There are solutions to this, of course. One can get network cards that have a BIOS that is aware of a diskless boot protocol developed to support diskless boots in unix workstations that can bootstrap both the kernel and the required device drivers with no disk at all. Some of the ways of obtaining the requisite hardware will be indicated in the Hardware appendix. This, however, is a bit tricky and there is a different way, I wouldn't quite say a better way, that is "directly" supported on a standard PC with over the counter parts.

That is to boot from a *floppy drive* (which is very cheap, generally costing less than $20) and use a standard cheap network card, but skip the hard disk altogether. Because this is the cheapest and most robust solution, this is the one we will develop in detail below. Let's start with the new and improved hardware list:

- A floppy drive.
- A generic SVGA card (I usually get $30 S3-Virge cards)
- Your NIC's

where again you may or may not be able to skip the SVGA card (depending on how hard you want to work and whether you want independent access to the nodes).

It is just about this moment of your life that you should pause and read the Diskless Howto, which is reasonably current and does a far more detailed job of describing diskless operation than I'm going to give you here. I will just outline the key elements:

- Build a boot floppy with the kernel and required network support for your distribution. The floppy will likely be a lilo floppy, and will have a whole little set of parameters that are to be passed to the kernel being booted. These parameters tell the kernel who the system is supposed to be (basically permitting it to configure its primary network interface) and where it should look for its root directory.

- On the server, build a root directory for the node and export it to the node. There is a huge range of ways to go about this. Some give every node a very large independent root. Others share one root among all nodes. Still others give each node a small independent root, but mount the "big" directories (typically /usr and /home) from a common server export.

- Develop a way of "cloning" the node files or directories on the server. Again, there are some very clever ways of going about doing this, from the simple but wasteful to the trancendentally clever and cheap but awesomely complex.

Note that this approach gives you essentially *all the scaling advantages* that the best method. You have a script titled something like "makenode" that you run on the server. It either clones a root and modifies the requisite files inside or clones the requisite files on a common root. It builds you a boot floppy, either customized to the node or generic. You pop the boot floppy into the floppy drive, power it up, and – Instant Node.

Not only that, but if a node goes down, a replacement can be brought up by popping the aforementioned floppy into the floppy drive and booting. There are also *loads* of Clever Tricks$_{\mathrm{tm}}$ that one can play – a system with a hard drive running WinXX by day can be rebooted into a beowulf node by night by popping in the floppy. A diskless WinXX node can be built (not that anyone sane would ever want one) by installing a diskless linux node and running e.g. VMware. To say there is no backup burden is an understatement – the nodes have no disks to back up and the server either contains a single image (with a handful of node specific files) to be backed up or a single image that is cloned to make the nodes that must be backed up – the script can reconstruct the node roots from this one image.

With all of this going for it, why is this method number *two*?

For two reasons. First of all, as you'll discover when you attempt to set it up, it *is* a bit tricky. You really need to know what you are doing to make diskless operation work. I used a diskless boot to "bootstrap" a cloning install to disk for our beowulf (back when I was still using slackware) and it took a lot of work and learning for me to do, in spite of my having run Sun SLC's and ELC's diskless for years. The Suns were easy in comparison, as they had boot proms that knew how to boot diskless *a priori*.

Second, your diskless system will probably have no swap, and will need a lot of memory to ensure stable, fast operation. In fact, you'll likely need to spend *most* of what you saved on a hard disk on extra memory unless you were already planning to build a node with a lot of memory (more than 128 MB, say). I'd advise adding at least 64 MB more than you planned on just to hold the operating system and your code. Linux is smart – it will use this extra pad of memory for buffering and caching libraries and disk files and so forth and greatly reduce the impact of mounting everything from NFS – and with or without this if you *ever* exhaust physical memory your system will die the aforementioned ugly death, probably right in the middle of your calculation.

Note that I do *not* advise turning extra memory into a ramdisk and going through some arcane ritual to load it with a large root. Linux turns it into a "ramdisk" for you, in all the senses that matter, and does a far more optimal job than you are likely to be able to do, while also being able to use the memory in other ways (like to avoid running out) in the event that an application *needs* it for a short time.

This wraps up our general discussion of beowulf installation approaches, in the order that you are likely to tackle them as a novice. Clearly there are lots of things to learn, and the best way to learn them is by doing. I rather wish that a lot of these were fully encapsulated in scripts and so forth for the novice, but thus far this hasn't happened. I make a stab at it in a few cases in the Software Appendix, but in other cases we all await contributions.

This by no means exhausts our need for clever tricks or advice on how to configure things. The next chapter is full of some (I hope) sound and sensible advice on the best way to set up certain aspects of the networking and so forth. First, however, we promised to address maintenance.

11.5 Beowulf Maintenance

In the preceding section we saw how our maintenance requirements can vary considerably depending on how carefully we automated our installation and how scalable (identical) our nodes are. If one sets the nodes up using either of the last two methods, they basically should not need to be backed up at all[21]. If one builds the nodes by hand, one probably will need to back them up.

The server node, of course, should be *carefully* backed up and indeed should likely have backup storage attached to match the disk it serves. Any decent book on linux or unix systems adminstration will tell you how to go about backing up space and I won't say any more of it. Once it is set up and automated, though, all you ever do is change an occasional tape.

As is the case with *any* network of computers, you will need to take care to monitor both systems messages (see suggestions below for centralizing this) and things like system load, CPU temperature, available memory and so forth.

[21] I do find it useful to spin off copies of e.g. their ssh keys and perhaps their /etc/X11/XF86Config (if any) to simplify a reinstallation, but this can be done just once when the node is created.

There are several very nice tools for the latter discussed in the Appendix on software (with URL's to download sites). If your beowulf or cluster is being used by lots of groups, you may need to referee demands for resources. Finally, you will likely have to teach some of these groups how to write parallel code for beowulfs. Giving them this book (and a handful of others) is a good first step, but of course there will be a need to do a bit more in some cases.

That's really about it. The nodes are "instantly" replaceable or re-installable, and as long as you protect and manage the head node like you would any server or workstation, the whole thing should just tick right along. Occasionally things will break in ways you don't understand at all – running a particular application will crash the system, or your network will fall apart.

That's what the beowulf list is for. So my parting advice on maintenance is to *join this list* at www.beowulf.org if you haven't already. I'll be there, and so will a lot of people who know far more than I about beowulfery and linux in general. Collectively, we know more than anybody! Chances are pretty good that we can help, and if we can't at least you'll know that the problem you are facing is *real*.

As you can see, beowulfs take a fair amount of work to plan. They take less work (once you've learned what you're doing) to build. Properly built, they take almost no work at all to maintain. Fix or replace the hardware when it breaks, upgrade the operating systems and tools periodically, maintain your server node and that's it. I don't spend an hour a week on maintaining my home beowulf, and on average spend little more on the one at Duke (mostly on hardware maintenance, actually – Grrr).

Chapter 12

Tools and Tricks

There are lots of ways, as time passes, to reduce the time you need to spend maintaining the cluster. Most of this one typically learns from experience, and as time passes most systems administrators accumulate a sort of “toolbox” of scripts or clever tricks that can be used to significantly reduce their work load while making their network ever more stable. This way they look good to their users while increasing the amount of time they have available to play Quake or listen to music or write the next killer app.

I’ve been doing Unix systems administration and systems programming (in addition to physics and all that) for way too long now (since 1985 or 1986 or thereabouts). I have therefore accumulated my share of these tools and have built up my own set of biases about the “right” way to do a lot of things. The tools are far from static in time – some things that were “right” or “clever” ten years ago are very wrong now. Also, from time to time I learn of something really clever from other folks that I never would have thought of on my own (for all my experience). Systems folks talk and share, which is a good thing as the terrain upon which they play and work is amazingly complex and so there are always things to be learned for the first time or relearned better.

This particular chapter is devoted to passing on a few little bits of systems management wisdom, mostly beowulf specific ones. To an experienced administrator, some of them may seem obvious (or even wrong, as systems persons don’t always agree about what is right). So take them with a grain of salt; try them out, and if they suit you feel free to adopt them. A number of these ideas actually have been discussed on the beowulf list, and I believe that what I present in those cases is more or less the consensus view of the best way to proceed, although as usual it is my fault if I have failed, not the list’s.

Let’s begin with something simple like the node naming and numbering scheme. It is a general consensus that it is advisable to use a *simple* naming and numbering scheme for your nodes. I tend to use things like b[1-N] (because I tend to work with smallish N). A good solution is to have a hosts table something like:

```
127.0.0.1          localhost.localdomain localhost   # Loopback
xxx.xxx.xxx.xxx    mywulf.lan.dom.org                # Outside address
192.168.1.1        bhead                             # Head node/server/gateway
192.168.1.101      b1                                # First node
192.168.1.102      b2                                # Second node
...
```

where I've assumed that this is a *true* beowulf with a head node (that doubles as a server and a gateway) that lives both on your organization's LAN (and xxx.xxx.xxx.xxx is its first address on e.g. eth0) and on the private LAN for the beowulf itself (the 192.168.1.1 "bhead" address on e.g. eth1). I'm also presuming, possibly foolishly, that you know how to set up appropriate routes for these two network addresses. There are HOWTO's to help you out for all this; use them.

A warning for the tyro's: *Do not use* the "0" and "255" network addresses for the most significant (rightmost) byte of an IP address. That is, only use 192.168.1.1 to 192.168.1.254 (at most) for host addresses in your internal LAN (or on your organizational LAN, for that matter). The zero addresses for higher bytes are ok to use (so 192.168.0.1 or 192.168.255.1 are OK), although I tend to avoid the zero address(es) because then 1 is the first address (which makes sense) and have never had so many hosts as to need the 255 address block.

The reason for this is that both 0 and 255 can function as broadcast addresses and tend to always match (or fail to match) wildcard addresses. It is also common enough to "reserve" certain blocks of addresses for particular functions. For example Duke likes to put routers on the X.X.X.250 address of any given subnet, so that it is always easy to guess how to set up routes on a new host. They similarly put the campus nameservers on the X.X.250.[1,2...] addresses so that nameservice can be configured easily. In many organizations X.X.X.1 is reserved for the primary server (and sometimes router) for a LAN. In the example above, I effectively reserved 192.168.1.[101-254] for nodes and the lower addresses for servers, head nodes, printers, or whatever.

The point is that a bit of thought and organization of your LAN IP space now can make life relatively easy later, as if nothing else it will be much easier to remember and implement consistently than mixing nodes, servers, printers, routers into the IP tables in first come first serve order. *If* you ever need to subnet your network (install routers between blocks of addresses) those address blocks will need to have a common netmask, as well, which argues for assigning blocks with boundaries that represent multiples of powers of two if you think that there is any chance of your doing so in the future[1].

If one wishes to have "vanity names" in addition to simple node names in the case of a NOW-style cluster, one can add aliases to a more normal hosts table:

[1]From this point of view, 101 is a foolish choice – I should start the beowulf at 192.168.1.128, 192.168.1.129, ... (for example) so that I can identify all "node" addresses by a mask on the highest bit. True enough, but I don't need to subnet and it's certainly easier to guess the IP number of b57 if one starts on 101.

```
127.0.0.1           localhost.localdomain localhost    # Loopback
xxx.xxx.xxx.xxx mywulf.lan.dom.org mywulf bhead  # server address
xxx.xxx.xxx.yyy toady.lan.dom.org toady b1            # First node
xxx.xxx.xxx.zzz  froggy.lan.dom.org froggy b2         # Second node
...
```

where mywulf, toady and froggy are all names of workstations (including a "server" workstation, in the case of mywulf) that double as beowulf nodes with names bhead, b1, b2 and so forth. In this case it is more difficult to "guess" what the IP number of b2 is from its name (as zzz may not be sequential to yyy) but one still identify nodes with a simple alias scheme.

If one has more than a few hundred nodes (lucky you!) then you'll have to extend this a bit and perhaps use a[1-100], b[1-100], c[1-100] on 192.168.1.[128-227], 192.168.2.[128-227], 192.168.3.[128-227], and so forth (to facilitate building netmasks, again). In this case to achieve anything like efficiency you'll almost certainly need a relatively complicated and expensive networking topology, high-performance routers, expensive switches or the like. At least 192.168.x.x has plenty of addresses to play with, though, (and 10.x.x.x even more!) so all of this is workable for pretty much any scale one might reasonably be able to conceive constructing a beowulf or cluster.

Now, why do we bother to arrange things like this, with a simple name and mnemonic numbering scheme? For many reasons. For example, it is now very simple to write a script to do all sorts of things on each node, one at a time. Examples of scripts like this in /bin/sh and perl are given in the beowulf software appendix. For another, one doesn't have to remember that mywulf, toady, froggy, and salamander are all beowulf hosts but that cobra, krait, mamba and newt are not. One can also tell Sally to use b[1-10] for a calculation while Tommy uses b[11-20], without having to tell Sally or Tommy just which hosts (by name) those are.

This latter idea extends to both setting up virtual parallel supercomputers within PVM or MPI or to configuring a cluster/LAN monitoring tool (one or two of which are also included in the software appendix). It's easy enough to work with ranges of either name or address space, but difficult to work with unique names and disconnected addresses.

If you've built a "true beowulf" with a real head node that functions as a gateway, it is also probably sensible to set this head node up to do IP masquerading or forwarding for the nodes (which is *very* easy to do in 2.2.x and higher kernels). Here is a place that some beowulf purists might easily disagree. A true beowulf built with lightweight nodes, one can argue, is a place where one "never" needs to login to a particular node at all, let alone access the outside world from that node.

My reasoning here (which I will stand behind) is that Murphy's Law makes it inevitable that one day one will want or need to login to a node, and even to login from one node to another node or to connect back to another workstation or resource in the outer world. That's why I recommend making the nodes "fat" as far as resources are concerned. It takes negligibly longer to install a node with the kitchen sink in available applications and resources, especially if one installs them in an NFS mount.

If one's favorite editors, xterms, debuggers, perl, and all the rest are all instantly available, the day you need them to cobble together some sort of "emergency" script designed to save your bacon you won't be trying to find some way of getting them onto a partially broken system that could die any minute and leave you with nothing. If you like, it makes hacking much easier, and any long-term system administrator knows that hacking a short term solution to an immediate problem may not be elegant but by damn it's going to be necessary, sooner or later. However, it also unleashes your creativity and capabilities for elegant and clever solutions to certain problems – if you need to update a particular directory tree on all your nodes, perhaps wget from a communal website is actually easier than messing with rsync and permissions, but this won't help you if wget isn't installed on the nodes and able to reach the external website.

With all that said, in a true beowulf one *would* normally require users or the administrator (you) to login to nodes either at the console (if there is a console switch of some sort) or over the network after logging into the head node (or "a" head node in cases where there is more than one) first. I can certainly imagine needing or wanting to get to say, my desktop workstation from a node, though, or even to a website as the wget example above suggests.

It is to facilitate this sort of thing that I made the installation of ssh a "mandatory" part of at least *my* recipe for a beowulf. Inside an "isolated" true beowulf (that might not "need" it for security) ssh manages forwarding of all sorts of network connections far better than rsh.

For example, starting at your desktop console in your organization LAN, if one ssh's into the head and then ssh's onto a node one can run X applications on the node and have the display automatically be set and forwarded back to your originating X console, which is rather awesome.

One can do even better. Both rsh and ssh have this nifty property that they check the name by which they are invoked, and if it isn't rsh (or ssh) it presumes that one is trying to run rsh or ssh to the host with the name by which the binary was invoked. The usual way to set this up is with a symbolic link, and Sun in particular had a standard directory and script for building symlinks for your organization so that rlogins or rsh's to systems within the organization could always be done just by the name of the machine.

Unfortunately, this hasn't been widely adopted within the common linux distributions, but the trick still works for both rsh and ssh (where we only care about the latter). In the appendix is a perl script for building a hostname symlink directory (historically, /usr/hosts) that contains, for example, a symlink from /usr/bin/ssh to e.g. /usr/hosts/mywulf. If /usr/hosts is on your path, then executing "mywulf" will log you into mywulf. Executing "mywulf xterm" will start an xterm on mywulf that should pop up on your current X console.

This permits a number of fabulously clever tricks. Presuming that mywulf has a similar /usr/hosts with symlinks for all the nodes, then executing "mywulf b1 xterm" on your desktop will crank up an xterm *on b1* that displays on your current X console, forwarding through all the intermediate connections transparently even if IP forwarding per se is off. The connection is even bidirectionally encrypted in the event that you need to type any passwords into the xterm. It is fairly simple to give b1's root permission to execute root-based GUI tools on your console, if you ever need to!

Another important thing to consider when setting up your beowulf is node logging. I advise that your nodes do run syslogd – there is too much that can happen that you'd want to know about. I'd also suggest that you do *not* log at all to local files, e.g. /var/log/messages and so forth that are the default. Log only over the network to your head node or a similar auxiliary node inside or outside the network. That effectively eliminates the need to backup /var on your nodes, and also significantly reduces the risk that a cracker can erase their tracks, if you defend the logging node even more carefully than you defend regular nodes or workstations.

This is by no means all of the clever tricks one can come up with for managing or operating a beowulf. Browsing the beowulf list archives will turn up many more. There is also a need for a lot more tricks to be contributed, as there is evidence that a lot of the tricks are invented by three or four or ten different people on the list independently, and sometimes one of those efforts is far better than the rest. The beowulf list facilitates a genetic optimization process, but this process works best when information flows in and out and can be compared and sorted and recovered.

Chapter 13

The Food Chain: Recycling your Beowulf

One lovely thing about cluster computing, including beowulfery per se is that there is a natural life cycle for cluster compute nodes. Let us meditate upon this life cycle for a moment.

Your grant is approved, your company agrees: You can Build a Beowulf. You collect quotes, select near-bleeding edge hardware, and put the whole thing together. It works! You do all sorts of fabulous research, or invent a new drug, or solve some critical problem, over the next two or three years.

Suddenly your once-new cluster looks pretty shabby. Ten percent or so of the nodes have given up the ghost altogether and been cannibilized for parts or been repaired at modest expense. Worse, Moore's Law has continued its inexorable march and it is getting hard to find nodes as *slow and ill equipped with memory* as your are any more, even for a paltry $500 each.

What do you do?

Well, an obvious thing to do is to buy shiny new nodes from *current* technology, and replace your old cluster with a new one some eight times faster at equivalent cost, but that leaves one with the problem of what to do with all the old nodes.

Welcome to the food chain. As systems age out (in any LAN or cluster environment) they gradually "lose value" compared to current technology, because

- Their reliability is diminished. Hardware failure starts costing administrative time and loss of productivity, especially if the cluster runs tightly coupled code.
- Their warranties (even extended warranties) expire, so you have to start paying out of pocket to fix them.
- Moore's Law dictates that spending a mere $100 to fix a three year old node is less cost-effective than using that $100 to pay for part of a (six to eight times faster) replacement.

- The overhead costs (power, cooling, space, networking, human management) for nodes scale more strongly with the *number* of nodes than with their speed and power. A 2.4 GHz Intel-based node might consume twice the electricity of a 400 MHz Intel-based node (to take a current snapshot that will of course require retranslation as these numbers advance) but requires 1/6 the space, 1/3 the TOTAL power and cooling, and 1/6 the management effort of the six 400 MHz nodes it might replace.

- Amdahl's law (see chapter 4) *usually* favors faster nodes to get more work done. One node at 2.4 GHz will generally complete work *more* than six times faster than six 400 MHz nodes. At *best* the six nodes would be equally fast, or nearly so.

Consideration of the above cruel facts may, in fact, convince you that it is better to upgrade your cluster *more* often than every three years. A lot of folks (myself included) try to arrange to upgrade their clusters once a year, with an explicit line item in each year's budget for a new set of nodes based on the technology du jour, skimming along near the crest of Moore's law instead of being lifted up to the top of the wave every three years only to wipe out in the troughs in between.

A totally dispassionate review of the Total Cost of Ownership (TCO) of the nodes in an associated Cost-Benefit Analysis (CBA) might well dictate *throwing the nodes away* every twelve to eighteen months rather than operating them until they die of old age. After this period, new technology is typically roughly 2x faster at equivalent cost, the overhead for operating the older nodes is 2x as great (per unit of work done), and the *human* cost of waiting for (presumably valuable) work to complete is often far greater than any of the hardware or operational costs. I have seen Real Live CBA's that prove this to be the case in at least some environments.

However, the proof depends to some extent upon the assumptions made (to include the infrastructure costs or not, to include the cost of the human time spent waiting for results or not). *Given* a set of assumptions and an assignment of costs and benefits, I can do no better than quote Dr. Josip Loncaric, a venerable and respected beowulfer[1]:

> Picking the best hardware replacement interval is an analytically solvable problem. Assuming that performance per $ doubles every N months, the most cost effective policy is to buy replacements whenever you can get 4.92155 times the performance for the same money. The Moore's law says that N=18, so the best replacement interval works out to be 3.44867 years. Using intervals of 3-4 years is almost as good.

This is the general view – most people view three years plus to be the ideal replacement cycle, and as Josip points out this is analytically justifiable. Note

[1] From private correspondance, June 5, 2001.

that this does *not* mean that replacing your cluster every three years is ideal – in general it will usually be better to replace 1/4-1/3 of your cluster (all three year old machines) every year, not replace the whole thing every three years. However, an *equally* good argument for a much shorter replacement cycle has been sent to me by a very competent list person who accounts for things like hardware reliability and so forth ignored by Josip. As always in cluster engineering, your mileage may vary according to your particular needs and cost/benefit landscape.

This still leaves one with the question of what to do with all the nodes one accumulates as they gradually age out, whether they age out in one year or five. The following are some very generic suggestions:

- Turn 1-3 year old nodes into desktops within your organization. Since one often buys relatively advanced nodes, they are likely to be strong enough to make good desktops (possibly enhanced with e.g. sound cards and CD drives) even when they are too slow to be terribly productive on your mail problem. This is the "food chain" – passing systems down in the organization until they become worthless to even the least demanding user.

- Create a hierarchy of clusters. Even older nodes can be useful for some problems, e.g. embarrassingly parallel projects with infinite time requirements (like my own research, so I know that projects like this exist). More or less unsupported, older nodes can often run for years doing useful work, even if it isn't *your* useful work. The downside of this is the power, cooling, space and network infrastructure the nodes consume. It costs roughly \$100 in power and cooling per year per (presumed 100W) node at \$0.08/KW-hr. A \$500 node will cost \$500 *more* in power over a five year lifetime. By the third or fourth year one reaches break-even on spending the power money alone for six nodes on a single node with six times the speed and less than six times the power requirement. However, in some cases *you* pay for new nodes, while power and AC are "free" (paid for by somebody else). Just one example of nonlinear cost profiles and how they distort decision making...

- Donate older nodes to organizations that can use their remaining lifetime profitably. For example, schools are often desperate for computers, and a three year old node (with a bit of updating) may be far better than what they have. Ditto for a number of non-profit entities. Schools may even be happy to take an entire beowulf, intact, so that they can use it to teach beowulfery! This is "like" passing it down the food chain within your organization, only passing it down to a different and poorer chain altogether. Sometimes realizing a tax deduction or other benefit in the meantime.

- Eventually, a node becomes junk. As in, it isn't worth plugging into a wall by anybody, even somebody poor and compute-power deprived. Or it is broken, and not worth fixing. Nodes in this state need to *really* be recyled, and not just thrown in a landfill.

Note well that computers contain a variety of toxic materiels. There is typically mercury in the little battery that backs up the bios. There is arsenic in the doped silicon in the IC wafers. There may be lead, cadmium and a number of other heavy metals used in various sub-assemblies. Computers also contain some valuable metals. There is gold on the contacts, for example, and plenty of copper everywhere.

There are good sides and bad sides to all of this. Node "recycling" often involves third world child labor and toxic materials (such as mercury) to extract the gold, and frequently ignores the rest of the toxic metals that build up whereever they ultimately dispose of the parts once the gold is mined out of it. We don't have the technology to disassemble nodes into reusable micro components, and even the reuseable macro components (such as the case and power supply, the drives, and so forth) tend not to be reusable for more than three to five years before they no longer work with current technology at all.

It doesn't do any good to recycle nodes "properly" where properly means sending them off to India to provide short term jobs and a toxic future for small Indian children. However, dumping them in landfills here isn't terribly wise either. Perhaps the best approach is to recycle the mercury-laden components (the battery) by hand, and landfill the rest, accepting that the arsenic and so forth will eventually show up in the water table. I'd be happy to hear better suggestions as this document reaches more people, and will cheerfully update this chapter as better ideas emerge. rgb@phy.duke.edu, people.

Chapter 14

Beowulfs Made to Order: Turnkey Vendors

As beowulfs have become more and more common, a number of companies have emerged that provide either hardware specifically tailored for high-end beowulf builders in universities or government labs or full turnkey "ready-to-run" beowulfs that are delivered pre-configured to operate and work on the problem(s) of the buyer's choice when it is first turned on.

14.1 Guidelines for Turnkey Vendor Submissions

If you are a turnkey vendor and are reading this chapter to see if you are there yet, well, probably not (or you'd be in the table of contents). However, you could be! Write a section describing your company and its offerings and send it to me and I'll consider adding it.

The document must be either plaintext or written as a latex
section{}. It should process out to strictly less or equal to three pages processed pages total and should *not* be written as "advertising copy". Rather it should focus on your company's technical strengths and what you can offer your customers as added value – technical consulting, programming support, special tools, great prices, whatever. No specific product prices permitted, but it is ok to indicate the general price range your offerings occupy. Contact information (e.g. a Web URI and a phone number) is strongly encouraged (that's where you can show or tell potential customers the detailed prices).

Figures or diagrams must at the moment be embedded postscript (EPS) only. Eventually I'll get set up to manage e.g. GIF inclusions for the web version of the book, but I'm not yet.

I reserve the right to reject or editorially modify any submissions. If I want to modify it, you of course will get a chance to accept my revisions before I actually publish it.

I am *not* selling space here, but I'm also not above accepting attempted bribes in the form of vendor T shirts for my boys (current ages 5, 10 and 13). I also am always open to donations of hardware – I like to try things before I write about them, and the more complete my hardware collection is the better I can compare and contrast the differences.

Part V

Beowulfs Everywhere

Chapter 15

Beowulfs in Business

Chapter 16

Beowulfs in Schools

Chapter 17

Beowulfs in Government

Chapter 18

Beowulfs in Developing Countries

One of the beauties of beowulfery is that *anybody* can afford supercomputing when the supercomputer is made out of off-the-shelf components. Over the last several decades, supercomputers have generally been export-controlled by the fully developed countries as a weapon. This has engendered a number of "interesting" discussions on the beowulf list over the years.

The rationale for restricting supercomputers as a weapon has always been that supercomputers can be used, in principle, to design a nuclear device and to model explosions for different designs without the need to build an actual device. The interesting thing is that this restriction was held from the early 1970's through the end of the millenium. During that time, a "supercomputer" went from a speed of a few million floating point operations per second through billions to trillions. We have now reached the point where personal digital assistants have the computing capacity and speed of the original restricted supercomputers. Computers such as the half-obsolete laptop on which I am writing this execute a billion instructions per second and would have been highly restricted technology a decade ago.

Add to this mix cluster computing methodologies that combine hundreds to thousands of of GFLOPs into an aggregate power of teraflops, at a cost of perhaps $0.50 per FLOP and (rapidly) falling. *Any country on the planet* can now build a supercomputer out of commodity parts, for good or for ill, capable of simulating a nuclear blast or doing anything else that one might wish to restrict.

It is difficult to judge whether or not these concerns have ever had any real validity. Of course, being a physicist, I never let a little thing like difficulty stop me. In my own personal opinion, the export restrictions haven't had the slightest effect on the weapons design process in any country, nuclear or not. Nuclear bombs have never been particularly difficult to design (remember that they were originally built with early 1940's technology!). The issue has generally not been how good a bomb one can design or modeling thermonuclear blasts, but whether one can build one at all, even from a time-tested design. In a nutshell, whether or not one could lay hands on plutonium or the appropriate isotopes of uranium.

In the meantime, restricting exports of supercomputers to only those countries already in possession of nuclear bombs or capable of managing the diplomacy required to certify their use in specific companies and applications had a huge, negative impact on the technological development of countries that *didn't* have nuclear bombs already or the right diplomatic or corporate pull. Engineering disciplines of all sorts (not just nuclear engineering) rely on advanced computing resources – aerospace engineering, chemical engineering, computer engineering, mechanical engineering, all rely heavily on visualization, finite element analysis, simulation and other tasks in the general arena of high performance computing.

Now, was this repeated extension of the definition of a "restricted supercomputer" *really* a matter of national security and bomb design, or was it (and is it today) a way of perpetuating the control certain large industrial concerns had over the computing resources upon which their competitive advantage is based? A proper cynic might conclude that both are true to some degree, but restricting the development of nuclear bombs alone is hardly credible in a world where we have increasing evidence that *any* country with the will and the plutonium (such as North Korea, India, Pakistan, at the time of this writing) easily *built* nuclear devices as soon as they had the plutonium, and only a lack of plutonium and a war stopped Iraq.

In any event, this amusing little bit of political cynicism aside, the playing field is now considerably leveled, and is unlikely to ever again become as uneven as it has been for the last fifty years.

I have personally built a cluster supercomputer in my home that has between eight and ten Intel and AMD CPUs in the range between (currently) 400 MHz and around 2 GHz (that is, half the cluster is really semi-obsolete and none of it is bleeding edge current). Together they easily cumulate to more than a GFLOP, making it a "restricted armament" as of a few short years ago.

The total cost of all the systems involved is on the order of four or five thousand dollars spent over five or six years. Spending five thousand dollars all at once I could easily afford eight to ten 2.4 GHz CPUs and quite a few aggregate GFLOPS (by whatever measure you choose to use), and this is chickenfeed spent on a *home beowulf*.

Using even *this* resource in clever off-the-shelf ways, I'm confident that I could do anything from design basic nuclear devices to model/simulate those nuclear devices in action, with the biggest problem being the creation of the software required from general sources and initializing it with the right data, not any lack of power of the computers. All the components of this compute cluster are readily available in any country in the world and are impossible to restrict. Export restrictions may or may not be dead, but they are certainly moot.

At this point any country in the world can afford beowulf-style supercomputing – a bunch of cheap CPUs strung together on a network switch as good as one needs for the task or can afford. And nearly every country in the world *does* build beowulfs. I've helped people on and off of the beowulf list seeking to build clusters in India, Korea, Argentina, Brazil, Mexico and elsewhere. Some of these clusters were associated with Universities. Others were being built by hobbyists, or people associated with small businesses or research groups.

"Armed"[1] with a cluster costing a few thousand dollars (and up), even a small school in a relatively poor country can afford to teach high performance computing – design, management, operation, programming – to prepare a future generation of systems managers and engineers to support their country's technological infrastructure and growth! Those trained programmers and managers, in turn, can run beowulf-style clusters in small businesses and for the first time enable them to tackle designs and concepts limited only by their imagination and the quality of their programmers and scientists, not by a lack of the raw FLOPS required for their imagination to become reality in a timely and competitive way. Compute clusters can also nucleate other infrastructure developments both public and private.

In the best Darwinian tradition, those companies that succeed in using these new resources in clever and profitable ways will fund further growth and development. Suddenly even quite small "start up" companies in any country in the world have at least a snowball's chance in hell of making the big time, at least if their primary obstacle in the past has been access to high performance compute resources.

In this country, I've watched beowulf-style compute clusters literally explode in "popularity" (measured by the number of individuals and research groups who use such a resource as a key component of their work). At Duke alone, ten years ago I was just starting to use PVM and workstation networks as a "supercomputer" upon which to do simulations in condensed matter physics. Today there are so many compute clusters in operation or being built that the administration is having trouble keeping track of the all and we're developing new models for cluster support at the institutional level. My own cluster resources have gone from a handful of systems, some of which belong to other people, to close to 100 CPUs, most of which I own, sharing a cluster facility in our building with *four other groups* all doing cluster computations of different sorts.

The same thing is happening on a global basis. The beowulf in particular and cluster computing in general are no longer in any sense rare – they are becoming the standard and are gradually driving more traditional supercomputer designs into increasingly small markets with increasingly specialized clients, characterized primary by the deep pockets necessary to own a "real" supercomputer. In terms of *price performance*, though, the beowulf model is unchallenged and likely to remain so.

Beowulfs in developing countries do encounter difficulties that we only rarely

[1] Sorry, with all of this talk of weapons design, which is likely to be the *last* thing on the mind of most developing countries, I couldn't resist the pun.

see here. Unstable electrical grids, import restrictions and duties, a lack of local infrastructure that we take for granted, theft, and graft, all make their local efforts more difficult and more expensive than they should be, but even so they remain far more affordable than the supercomputing alternatives and well within the means of most universities or businesses that might need them. As is the case here, the human cost of a supercomputing operation is very likely as large or larger than the hardware or infrastructure cost, at least if the supercomputer is a beowulf or other compute cluster.

Even with these difficulties, I expect the global explosion in COTS compute clusters to continue. Based on my personal experiences, this book (in particular, given that it is available online and free for personal use) is likely to help people all over the world get started in supercomputing, nucleating new science, new engineering, new development.

To those people, especially those in developing countries trying to overcome their own special problems with funding, with environment, with people, all I can say is welcome to beowulfery, my brothers and sisters! The journey will prove, in the end, worthwhile. Please let me know if I can help in any way.

Chapter 19

Beowulfs at Home

This is a subject near and dear to my heart as I've had a beowulf at home for several years now. Amazing as it may seem, it is now entirely possible to build a beowulf-class cluster at home, and in fact there are some very useful reasons to think about doing so.

It has become possible for several reasons:

- Computers have become almost unbelievably cheap. Very respectably configured Intel or Athlon based computers are available for as little as $750 (US) *with* a graphical head, Windows installed, and some useless gingerbread. If you can get them to keep Windows and give you more memory, a bigger disk, and perhaps a nicer monitor you can get a "head node" (which doubles as a general purpose desktop) for under $1000.
- Compute nodes, which need be little more than a case with a processor, some memory, and a network card (no, you don't need even a floppy disk if you have a PXE/bootp NIC) can cost as little as $500.
- Even a fast ethernet switch, which used to be quite costly, is now available for as little as $100 for 8 ports (quite enough for a small beowulf).
- All the components (running Linux) can serve *more than one purpose*. In my house, many nodes are also desktops. In this way I cover my wife and kids' desktops and *also* get to use their (mostly idle) CPUs over the network. Obviously this won't work with any systems running Windows, unless it is within e.g. VMware.

The most difficult single thing about setting up a home beowulf with nodes distributed in several rooms is likely to be the physical network. Houses that are properly wired with category five cabling and RJ45 sockets in all the rooms are relatively rare. However, this is compensated for by three things.

For one, wireless networks are now relatively cheap. A wireless access point is less than $200, and a wireless card for a PC or laptop less than $100. Wireless bandwidth isn't terribly exciting (I tend to get 2 Mbps unless I'm sitting on top of the wireless access point with my laptop) so you won't be able to run particularly fine grained code, but wireless is perfectly adequate for coarse grained code and embarrassingly parallel applications.

For another, having your house wired for a network by professional electricians isn't horribly expensive, either. Depending on how hard they have to work, it should cost on the order of $100 per pull from a reasonable central location (your "wiring closet" or wiring shelf, where you will locate your switch, a small patch panel, and a cable and/or DSL phone line for external broadband access) to various rooms. Most of this cost will be labor – it is only a little more expensive to pull 2-4 cat 5 cables than it is to pull one.

Having four rooms wired (with perhap 12 ports, three per room) and a terminating patch panel installed might cost around $500 or even less. It will one day increase the value of your house, and in the meantime it will make your life better in lots of ways. Go for it.

The final way to proceed is, in the best of beowulf traditions, to Do It Yourself. The wiring, that is. This is remarkably simple, and cuts out all the labor costs relative to a professional installation. The hardware costs are a few hundred dollars for cabling and termination and tools, and you can install as many lines as you need whereever you need them.

In my own house (what better testbed?) I have owned and operated a beowulf for years now. I've even written it up in an article for ;login, complete with pictures[1]. I used all three of these methods to network my house. I hired professionals to wire our attic as we had it remodelled into a home office. I stuck a wireless access point up there (and am working on my laptop in my living room via wireless as I write these words). I pulled wires myself into all the downstairs bedrooms, terminating them (and all the professionally installed wires) in an RJ45 patch panel. I pulled a category five phone line from our phone service up to the patch panel for DSL, and a cable up against the possibility that we might one day use cable instead of DSL (or might want to watch TV on a computer, as unlikely as that might seem).

I am therefore in a good position to tell you...

19.1 Everything You Wanted to Know about Home Networking but were Afraid to Ask

I'm going to keep this fairly simple. Do Not Try This if you are clueless about wires, electricity, following instructions, and using tools (including power tools). I'm going to assume that you can operate a saber saw, a drill, electric screwdriver, and so forth and hardly ever cut off an important limb or digit. I'm going to further assume that you understand that electricity likes to run inside conductors, hates to be shorted out, and (in higher voltages) would just love to

[1];login, The Magazine for Usenix and SAGE, August 2001, vol. 26, number 5. This is available online at: http://www.usenix.org/publications/login/ 2001-08/pdfs/brown.pdf

plunge to ground through your delicate and easily electrofried anatomy. The network wiring itself carries only trivial voltages and currents during normal operation, but who knows what Evil lurks in the hearts of walls, especially the walls of older houses? If you're careless or unlucky, you can cut right through household wiring while doing something as innocuous as cutting out a hole for a circuit box in your wall.

Then there are your local laws and codes, which may or may not be happy with rank amateurs installing their own wiring, even low voltage low current wiring.

So, before we begin, permit me to say *proceed at your own risk*. It isn't going to be *my* fault if you burn down your house, electrocute yourself, lop off a finger with a saw, or even follow all my instructions perfectly but end up with a network you can't quite get to operate. Tough. If you wish to avoid risk, pay a professional.

There. Now we can proceed.

If you are still reading, I presume that you're undaunted, ready to DIY, maybe even a bit cocky about your mechanical abilities. Good. It really isn't that hard, nor is it that dangerous, but there are plenty of Complete Idiots out there in the world who can manage to bollix anything up – and then sue you for it, as if their stupidity or plain old bad luck was your fault.

To install your own wiring, you'll need to begin with a trip to Home Depot, or Lowes, or you own local favorite hardware and wiring mecca. A shopping list of tools and parts you will need or find useful:

- A box or boxes of category 5e, data quality, unshielded twisted pair cable. A box typically contains 1000', often in your choice of garish colors (but grey or white will do fine). A box typically costs somewhere in the $50-100 range. Don't bother getting less than a full box – you'll find it useful for many things even if you only "need" 500' to run all the wires in your plan. You can donate any surplus to a local school or other charity.

- The usual range of tools for working with wire and walls – wire cutters, needlenose pliers, screwdrivers in various sizes (including a rechargable electric one if possible), a pocket knife, a rechargable drill, a rechargable saber saw with plaster, wood, plastic, metal blades, a utility knife, tape measure, level(s), hammer, and so on. If you are the DIY type, you probably own most of this already, and are happy to have an excuse to buy anything you are missing ("...but he SAID that I'd need a brand new metal lathe, honey...").

- A wire puller, at least 25' and maybe longer depending on the length of the runs you need to pull. This is basically a reel of retractible/extensible spring steel with a little hook on the end. You poke it down walls and up through floors, fasten wires onto the hook end, and "pull" them back. Also known as a "fishtape" because one spends a lot of moderately frustrating time "fishing" for holes, wires, passages through the mystery maze behind the drywall, and so forth.

- Some largish bits for your drill or brace-and-bit – 3/8”, 3/4”, 1”, $1\frac{1}{2}$” all might be useful to make holes big enough for one or more cables.

- A crimping tool. I have one that crimps both RJ11 (phone) and RJ45 (phone/data) ends, and can make my own phone or data cables on demand. Something to do with all the leftover wire.

- A cable tester. This is *essential*. An inexpensive cable tester typically consists of two pieces – one with a battery in it that plugs directly into one end of a connection, one with a bunch of little lights on it that plugs into the other end. It basically switches the juice onto each pair in turn, and if the pair is good, lights the little light for that pair. Without this tool, you'll never know if a failure is bad wire, bad wiring, a bad network card, a bad software driver for the network card. Even *with* this tool this may not be horribly clear, but you can at least eliminate the physical network from the list of possible problems with a high degree of confidence.

- If you plan to use some of the cables for phone extensions, get a phone circuit/polarity tester. Otherwise you've got a 50getting the polarity wrong, which probably won't break anything but isn't right. Probably being the operative word...the phone company just hates it if you mess up a phone circuit, and reversed polarity can be annoying if not dangerous.

- Circuit boxes to put in the walls. Here there are many very distinct choices to be made. One usually uses the same size boxes that one would use for e.g. 110V wall receptacles or wall switches. Such boxes can install between straps or against a wall stud (in new construction), latched into a hole in drywall (nifty little sideswing locks), snapped into a hole in drywall (spring loaded locks, screwed or nailed into a stud through a hole in drywall, or fastened/glued ONTO drywall from the outside (usually mated with cable channels that are also glued onto the wall on the outside). Note that code may also require conduits, separate grounds, supports inside walls, and anything else that code wants, if you can figure this out in your locale and care. Do Not Share a Box with Real Electricity In It.

- A variety of terminating devices. One can buy RJ45/data plugs or RJ11 phone plugs in bulk (boxes of 10, for example) or one at a time for about twice as much. There are face plates with one hole for a data port, two holes for data and (say) phone. There are plates with 3, 4, 6 and even 8 holes where you can install any combination you like of data, phone, cable, and even wires for speakers. I generally recommend pulling at least two data lines if you pull any at all, and three or four is better. A single cat 5 line will support up to four phones lines (four twisted pairs)[2], by the way – you don't need to buy or pull separate phone cables. If you wire the

[2]It may sound stupid for me to say this, but you can't carry both phone and data at the same time, so can't use the leftover wires in the phone cable to carry anything but more phones.

plugs correctly, you can also plug a phone line (RJ11) into a data (RJ45) socket and have it work (middle two pins). Consequently, you don't really *need* RJ11 sockets at all, even if you plan to use one of the lines for a phone extension.

- You may or may not want a patch panel for the location where all the wires are pulled *to*. If you're only pulling a few wires (say eight or less), you can maybe get by with 1-2 circuit boxes ganged together and 4-8 RJ45's installed in each. If you're pulling more than eight, I'd recommend a patch panel. There are a variety to choose from – rackmount, wall mount, shelf mount – and they cost in the ballpark of $60 for 12 ports. Do *not* plan to terminate the free ends of the wires with RJ45 plugs as this makes the installation difficult to label and test and debug.

- A cheap cable stripping tool. These are things that look like clips or clamps with an adjustable razor edge. The razor can be set so that you can cut through the insulating jacket on cat 5 cable without nicking the wires inside, which is a Very Good Thing to be able to do quickly.

This isn't as bad as it sounds. A lot of this is the hardware you will install (circuit boxes, plugs and faceplates, wire, patch panel). Some is tools you probably already have. Aside from that, you need the crimper, the stripper, the tester(s), and the puller (a total cost of maybe $120).

Next you need a plan. Central to the plan is your "wiring closet" – the place you're going to pull all the circuits back to. This location might be a closet, a shelf, a cupboard, the garage, a big box (like a circuit box) in a wall. Air conditioned space with an electrical receptacle handy is preferred as you'll want to locate the network switch there, maybe a wireless access point, maybe a DSL modem or cable modem or a gateway there, and all of these need power and generate heat.

It is nice to have a cable drop there (and to pull TV cable along with your network wiring if you're house doesn't already have it). It is nice to have BOTH a regular phone line AND a data phone line (if your service box, like mine, is already equipped with a splitter and separate wiring points). You may or may not need some degree of physical security there – at the very least keeping little fingers out (if you have children) is a good idea, or you may just plain need to lock up the equipment.

However, the *essential* feature of the space for home wiring is that you *must* be able to pull cables into and out of the space. This means that you need to be able to put a hole into the walls, the ceiling, the floor of this space and access SOME route to whereever you plan to put a plug. In most houses this means that the space will need to be in the top floor (where you can come down from the attic) or the bottom floor (where you can come up from basement or crawl space).

Some houses are nearly impossible to wire without removing a lot of drywall or at least cutting lots of holes in drywall and then patching them closed when you're done. If you live in such a house, you'll have to decide whether or not you want to mess with DIY wiring at all, as it won't be easy or pretty and I'm not going to cover drywall repair methodology herein so you're mostly on your own.

One solution for even this case that I will mention and that isn't *too* horrible is to use wiring channels and boxes that are glued or screwed onto the drywall from the outside. This leaves one stuck drilling holes through walls here and there to let cables through, but keeps one out of the drywall itself. Maybe if you are a bachelor...

Once this place is located and routes you can pull to all terminal locations established and put down on a plan, you pretty much just do it. Doing it consists of:

1. Drill holes in the floor/ceiling/walls as necessary to get a wire from the wiring closet...

2. fishtaped into the wall and...

3. up (down) into the attic (crawlspace)...

4. from where it is safely routed (in conduit, through holes in joists, stapled to joists with cable staples) over to the room being wired...

5. and pulled down (up) into the wall of the room...

6. out into that room, through the circuit box, with your fishtape.

Leave a generous hank of cable on both ends. Pull more than one cable at the same time to the same place whereever possible. You can't have too many cables – unused cables are spares, can serve as extra phone lines, and at worst cost a few dollars in "wasted" cable and parts if unused. You *can*, however, have too *few* cables, and pulling more cables to fix up some room when the rest of the project is done will be a real pain, easily avoided by overwiring now.

Strip an inch or so of jacket from the cable in the room. Using the (usually provided) punch tool, untwist and wire the ends *precisely* as indicated on the RJ45 plug. You will often see two alternative wirings (each with its color scheme) represented on the plugs. Use either one (A or B) but *use the same one on all plugs you wire, on both ends* or you'll make a big mess and nothing will work. I generally use A.

The idea is that all the little one pins have to be wired to all the other little one pins, two to two, and so forth. Mixing mixes up your signals and nothing will work.

When all the wires to that location are so wired, snap the RJ45's into the face plates and wire the whole thing into the box (careully pushing surplus wire up into the wall. Do the same thing on the wiring closet end, either terminating the wire (using the same scheme!) in a punchblock or in an ordinary RJ45 in a faceplate.

TEST THIS CIRCUIT now with your tester, and LABEL THIS CIRCUIT at both ends. You need to know that plug 3a (your middle son's room, top plug) is also plug 3a in your main patch panel and tested between those two points.

The hardest part of this whole procedure is probably fishing for the wires in the walls. Some walls are easy and you just drop in one hole and easily find the other down below. Other walls are full of insulation, of fire blocking, of other wires (dread electrical, phone, cable), of plumbing (DON'T mess with plumbing) or you turn out to be between studs four and five up above while the hole you cut in the drywall for the circuit box turns out to be between studes five and six...

Be careful, work systematically, test often.

19.2 The Rest of the Story

Once the wiring is in place, the rest is easy. Install nodes/desktops whereever you need them, hook them into the network (possibly using cables you made yourself out of the leftover wire and a box full of RJ45 crimp-on plugs), install linux including PVM and MPI and parallel tools and compilers, and have at it.

What can you do with it once you've built it? Well, for one thing, a home beowulf usually has a reason to exist as components alone. Each machine usually "belongs" somewhere and does several things. What you do with it along the lines of parallel computation depends on what you do and what you want to get out of it. I use mine for code development, as a testbed for parallel applications and management tools, and to a lesser extent as a production unit in my research. I don't need a lot of nodes or fast nodes or matched nodes for these purposes, but working at home frees my nodes at Duke for production and keeps buggy code off of important machines until it is pretty well debugged.

Chapter 20

Justifying a Beowulf

A *sine qua non* of beowulf engineering is paying for it. Few individuals buy beowulfs "out of pocket" (I'm one of the admittedly rather strange exceptions to this rule) and everybody else has to find the money for one. This money generally belongs to somebody else – the government, the university, the company – and one has to either write a grant proposal or a some sort of justification for the expenditure of money already in hand.

How can one justify purchasing a beowulf to people who in many cases haven't the foggiest of ideas about what a supercomputer is or how they work or what one might want to accomplish with one? In a really ideal universe, one would simply say "because without one I cannot get my work done and we all agree that that work is worthwhile" and be done with it, but alas this is often viewed as an unsatisfying answer by the administrative individuals charged with ensuring that valuable money is well-spent.

The key step to justifying a beowulf purchase is the gentle education of those individuals in charge of providing the money. These individuals are generally not stupid – they are merely ignorant, and ignorance is easily curable. This chapter is devoted to a rational but *very simple* explanation of the beowulf concept suitable for cutting and pasting into purchase justifications or grant proposals. It therefore departs somewhat from the wry or humorous tone of previous chapters[1].

Get out those figurative scissors, boys and girls. I'm writing the following section in order to use it *myself* in a matter of moments, as I too (outside of my autofunded home efforts) am a slave on the Wheel of Life, a bottom-feeder in the Aquarium of Academe dependent on food sprinkled by government hand, an OPM (Other People's Money) addict – in short, one who has to not infrequently explain to a variety of folks how and why I'm using their money wisely by following the Beowulf Way instead of just taking the easy way out and spending ten times as much on a Cray.

[1] Did you notice how I pointedly refrained from calling the administrative individuals something humorously pejorative such as "bean counters"?

20.1 Beowulf Description

A beowulf-class parallel supercomputer is a cost-effective way to acquire supercomputing resources to support a variety of kinds of numerical research, generally in physics, applied mathematics, or engineering but also in weather prediction, nuclear weapons design, quantum chemistry and in many other fields as well. Architecturally, a beowulf consists of some (variable) number of commodity, off the self (COTS) computers (called "nodes" or "compute nodes") to do the actual computations, one or more "head nodes" (COTS computers configured as user workstations and arranged to provide access and services to the nodes), and one or more interconnecting COTS networks so that the nodes can communicate with the head node(s) and each other. This collection of computers runs an open source operating system (generally linux, although there are a few exceptions) configured in such a way that the usage of all of these compute resources as a single entity performing parallel computations is facilitated.

The compute nodes are often (but not always) configured without any kind of monitor or keyboard; they are accessed only via the network. They generally will have a good deal of memory and relatively fast networking and central processing units (CPUs). They may or may not have local disk resources such as CD-ROM drives, hard disks, or floppies.

The head nodes will often have high-end graphics, large disk arrays, attached printers, scanners, or other peripherals and will provide shared access to disk space and peripherals to the sparsely equipped nodes. These head nodes will generally sit on desktops and can be viewed as the "console(s)" of the parallel supercomputer. They often function as visualization workstations, program development platforms and may provide network isolation functions to keep the supercomputing network secure and free from extraneous traffic.

The network plays an essential role in beowulf design. It allows work that is split up so that it can be executed in parallel on the compute nodes to be sent to those nodes (often from a session running on a head node). It is then used to provide data to those running tasks, and to allow those tasks to exchange information with tasks running on other nodes as required by the computation.

Some parallel computational tasks require a great deal of computation but not much communication. Beowulf design for these tasks will generally emphasize raw compute speed and large memory and spend relatively little on networking. Other tasks have a lot of communication and relatively little computation. Beowulf design for these tasks will generally emphasize the network; it is not unheard of to spend more on the networking being delivered to a node than is spent on the node itself.

Similarly, some computational tasks require that tasks be completed by the nodes in a very predictable, regular ("synchronous") way – every node needs to complete a subtask at some particular time so that they can all communicate and go on to the next subtask. Program design for this sort of task will be much simpler if all the nodes are the same and their performance homogeneous and predictable. Other computational tasks just do chunks of work that are more or less independent of other chunks and can be completed at any time and in any order. In this case it is much less important that the nodes and network be identical.

Either way, the goal of beowulf design is to provide an aggregation of (individually) rather inexpensive compute power to apply to advanced numerical tasks. Unlike a "dedicated function" parallel supercomputer such as a Cray T3E or IBM SP3, a beowulf is rarely purchased all at once and follows a much different depreciation and component reuse cycle.

For example, nodes can be purchased and added at any time (provided that sufficient networking resources exist or are also added to permit the new nodes to be interconnected. Old nodes that are no longer sufficiently state of the art to form adequate compute nodes can be cycled out and, with the addition of a monitor and keyboard, often can function for years longer as perfectly adequate desktops. The same is true of network components and even head node components. A beowulf design is emminently recyclable.

A beowulf is therefore rarely bought all at once and then "finished" in the sense that a commercial single-chassis supercomputer is purchased all at once and then used as a unit until it is retired. Nodes are added, nodes are removed, nodes are replaced or repaired, and the "supercomputer" carries on. New nodes for a single beowulf are often funded from several sources over many years; the Duke Physics department's primary beowulf, "brahma" (see http://www.phy.duke.edu/brahma) has been in existence for five years across two or three generations of hardware.

Nevertheless, in all important ways a beowulf is just as much a real supercomputer as its commercial brethren, only much, much cheaper and much, much longer lived. In particular, as an identifiable (often named) network entity, it must generally be treated as capital equipment rather than as a mere aggregation of desktop workstations, regardless of where its components are actually physically located. Otherwise one's design and purchase process is heavily distorted by the need to pay indirect costs on individual nodes while those node costs in aggregate vastly exceed the capital equipment threshold.

Chapter 21

Portable Beowulfs

21.1 Special Engineering Problems

For example, a discussion of using low wattage components e.g. laptop motherboards, IBM CPU motherboards, Transmeta CPU motherboards. Issue to be checked – is power savings due to idle mode or half speed mode or is it just using superior/cooler VLSI technology? Be nice to get somebody to write this (volunteer anyone?) who's actually built or is building one. Or get e.g. IBM or Transmeta to donate hardware so I can build one and write about it. Oooo, good idea. I'll have to look into this.

Basically, though, the things to be overcome in a portable are:

1. Low weight
2. Small volume
3. Minimal power draw
4. Minimal cooling
5. Shock Resistance
6. Configuration of "head node" and the network.

Probably aren't a lot of Gbps solutions out there, also.

21.2 Portable Example(s)

(as anybody gives me any).

Part VI

Contributed Articles and Chapters

Chapter 22

Contribution Template

Robert G. Brown
Duke University
rgb@phy.duke.edu
http://www.phy.duke.edu/ rgb

22.1 Contribution Remarks

This is a template contributed chapter. If you are interested in contributing material to this book, you can download the full book source from http://www.phy.duke.edu/ rgb/Beowulf/beowulf_book.php. This chapter is one of the sources. Make a copy of the template and write your article. Send it to me.

22.2 Contribution Rules

To avoid having to mess with complicated copyright and licensing issues, I will make them very simple. The book itself is and will remain Open Publication Licensed. That means that the book itself will remain free to anyone who wishes to use it for their own no-resale-for-profit purposes. Any work that is contributed must use the same license. That way I can freely include your work and everybody is happy. Note well the beverage clause! So far, this has garnered me a whole two drinks, but one never knows...

This leaves open what will happen regarding *non*-free publication. It is entirely possible that one day a publishing house will publish a paper copy of this book and pay me money for the privilege of doing so. Note that this will not affect the availability of the book online – it is just that a paper book is more suitable in many ways for a deskside reference or a classroom textbook, and besides, you can have it signed by the author and it works when your laptop batteries are out or you are off the network.

The "rule" for submissions/contributions is that if I sell the book for money, you can either:

- Let me include your contributed work and make money from it. Thank you very much! (Obviously I like this one best.)
- Let me include your contributed work but make a small donation to e.g. the Free Software Foundation or Care in your name out of the massive profits I thus derive.
- Ask that I not include your work. No problem. If I feel that the contributed material is really important to include, I'll rewrite the material in original prose.
- Possibly some other solution exists that the publisher will suggest. We'll play that by ear.

You don't have to choose now – as long as your contact information is correct I will ask you if/when the for-profit issue ever becomes real.

22.3 Editorial Rights

I reserve the right to edit any submissions and make corrections, omit material, or do minor reorganization as required to achieve a reasonably consistent look and feel, to any submitted work. If the revisions required are major, I'll either run them by you (the author) before accepting the material or more likely, ask you to rewrite according to some suggested guidelines.

I also (of course) reserve the right to reject any submissions for any reason or none at all. That is, rejection may not reflect badly on the quality of the submitted material but rather on it being too redundant with existing or planned material or on my having a really bad day and feeling hostile toward the world.

22.4 Solicited Material

I would be *happy* to get topical articles on specific narrow-focus areas of cluster computing such as (but not limited to): high-end network cards, comparative results of specific benchmarks, example code and/or programming techniques, specific parallel applications.

It is a really good idea to contact me before contributing a lot of material to be sure that I agree that the contribution is likely to be welcome.

Part VII

Conclusion and Appendices

Chapter 23

Conclusions: The Path to the Future

Appendix A

Beowulf Software: Libraries, Programs, Benchmarks

Appendix B

Beowulf Hardware: Computers, Networks, Switches

Appendix C

Beowulfery and Me: a Short Memoir

Let me tell you a little story – the story of how I came to be involved in beowulfery in the first place.

One's goal in a Monte Carlo calculation (at least the ones I do as a physicist) is to importance sample a Markov chain driven by a suitable stochastic selection rule. The result is built up from many, many "independent, identically distributed" samples generated by the Markov process[1]

However, the Markov process does not in general produce completely independent states for sampling in each step. In the problems of interest the number of steps that must be taken to generate one *independent* sample tends to increase with the size of the system studied (according to a scaling law of its own). It can take a *lot of samples* to generate high precision results for large systems which have the greatest relevance to the physical phenomena one is simulating. It can take a *very* large number of steps in the Markov process to generate this large number of independent samples.

By "large numbers" here I mean that at one point five or six year ago I ran the calculation on over 100 Sun Sparc 5's for nearly a year, continuously. This added up to well over 2 GFLOP-years of computation. I'd be running on them yet if it weren't for the fact that Solaris (2.3) proved utterly incapable of managing my calculation in the background and a foreground X session simultaneously and even more unfortunately, my background process usually "won". This irked the students at Duke University that these compute clusters technically "belonged" to[2]. So I got booted from the student clusters and immediately started to design

[1] Note that if you don't know what a Markov process is or how importance sampling Monte Carlo works, don't worry about it. Imagine me rolling lots and lots and *lots* of electronic dice and playing snakes and ladders. It's close enough.

[2] Narrow minded of them, don't you think? Do your computer science homework on time or advance the cause of science? Decisions, decisions. Actually, they rapidly learned to just reboot the systems when they discovered my job running, temporarily foiling my automated job spawner...

Duke's first beowulf, before I even knew that the beowulf project existed.

The design process was simple. SunOS might have worked as an operating system for the cluster – I'd used Sun's and PVM and expect for years at that point doing these calculations and with SunOS they ran nicely in the background without annoying users working at the console. Solaris was out; I *still* haven't forgiven Sun for taking an operating system that, really, was no worse than linux 2.0.x and transforming it into something hideous. Linux I'd used and had marvelled that it was so wonderfully functional - every bit as good as SunOS (which at the time was arguably the world's best operating system – I'm not dissing linux at all).

Intel Pentium Pro's had also just been introduced and really for the first time could compete with any of Sun or SGI or DEC's affordable workstations. So I added a linux-dual PPro cluster onto our next grant proposal. It was actually something of a relief to discover the beowulf project (via a visiting physicist from Drexel). I no longer had to wonder if linux was sufficiently stable to support a compute cluster as it had already done it.

However, I didn't have to worry *too* much, because my task was ECG. I had managed it for years using an unholy mix of submitting jobs one at a time on lots of hosts on our lan, then as a master-slave tasks (see below) via PVM, then (to avoid loading the campus network backbone, which I was far more worried about than loading Sparc 5's that had cycles to spare compared to my desktop SunOS Sparc 2 that could run my calculation in the background and X in the foreground without difficulty) via expect scripts and /bin/sh scripts. I knew that as long as I could construct a system that had *a* network, some reasonably fast floating point, and rsh that I could get useful work done.

Finally, the systems arrived, I popped linux 2.0.0 in Slackware onto them (2.0.0 was still so new that it crinkled when you rubbed it, but it *did* support SMP operation and I had bought dual PPro's exclusively) and, after a week or two of pounding my head against the hardware trying to get an Adaptec 2940 to work[3] I'd learned a lot about the kernel, had for the first time in my life managed to actually break a filesystem by (necessarily) shutting a system down without syncing it, had gotten pretty good at the Slackware floppy-based install, had learned to flash an Adaptec BIOS, had moved up several kernel revisions to 2.0.11 or so, and had a very small stack of dual PPro's on switched 100BT (which at that time was *very* expensive) running a parallel calculation.

I'd also joined about six linux lists, including at some point then or soon thereafter, the beowulf list. The beowulf list proved to be a godsend. There were *other* people out there who were using linux-based COTS computers to build supercomputer-class clusters. They were going through the same things I was. Making the network cards behave (some didn't). Deciding whether single CPU boxes were "better" or "worse" than dual CPU boxes (answer: it depends, as discussed in the chapter on bottlenecks). Learning to be envious of those who

[3]This was long ago when if you'd said "linux" to an Adaptec rep they'd have said "Huh?". Adaptec at the time had this nasty habit of changing the card BIOS without changing their revision number. I believe they still do.

had really big piles of systems with snazzy superfast networks like Myrinet (in spite of the fact that I, at least, didn't really need a superfast network).

The folks on the beowulf list have taught me, step by step, most of what I know about cluster computing. Sometimes the lessons were pleasant and fruitful, other times I got by skin blasted off by nuclear flames, but both ways I learned and eventually got to where I could contribute some of what I'd learned back to the list.

In case I haven't made it sufficiently clear yet, I am *not* a real computer scientist, I'm just a humble physicist. There are plenty of folks on the beowulf list who *are* real computer scientists, some of them Bell-prizewinning scientists at that (I've seen the one hanging on Don Becker's wall, for example). To all of them I am humbly grateful, and to them I dedicate this book[4].

[4]If nothing else, guys, maybe I can start answering certain questions with something like "download this book" and reduce list traffic by some fraction of my not inconsiderable expostulation.

Appendix D

Bibliography

Bibliography

[beowulf] See `http://www.beowulf.org` and links thereupon for a full description of the beowulf project, access to the beowulf mailing list, and more.

[Amdahl] Amdahl's law was first formulated by Gene Amdahl (working for IBM at the time) in 1967. It (and many other details of interest to a beowulf designer or parallel program designer) is discussed in detail in the following three works, among many others.

[Amalsi] G. S. Amalsi and A. Gottlieb, *Highly Parallel Computing* (2nd edition), Benjamin/Cummings, 1994.

[Foster] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995. Also see the online version of the book at Argonne National Labs, `http://www-unix.mcs.anl.gov/dbpp/`.

[Kumar] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing, Design and Analysis of Algorithms*, Benjamin/Cummings, 1994.

[lmbench] A microbenchmark toolset developed by Larry McVoy and Carl Staelin of Bitmover, Incorporated. GPL plus special restrictions. See `http://www.bitmover.com/lmbench/`.

[netperf] A network performance microbenchmark suite developed under the auspices of the Hewlett-Packard company. It was written by a number of people, starting with Rick Jones. Non-GPL open source license. See `http://www.netperf.org/`.

[cpu-rate] A crude tool for measuring "bogomflops" written by Robert G. Brown and adapted for this paper. GPL. See `http://www.phy.duke.edu/brahma`.

[Eden] The "Eden" beowulf consists of lucifer, abel, adam, eve, and sometimes caine and lilith. It lives in my home office and is used for prototyping and development.

[profiling] Robert G. Brown, *The Beowulf Design: COTS Parallel Clusters and Supercomputers*, tutorial presented for the Extreme Linux Track at the 1999 Linux Expo in Raleigh, NC. Linked to `http://www.phy.duke.edu/brahma`, along with several other introductory papers and tools of interest to beowulf developers.

[ATLAS] Automatically Tuned Linear Algebra Systems, developed by Jack Dongarra, et. al. at the Innovative Computing Laboratory of the University of Tennessee. Non-GPL open source license. See `http://www.netlib.org/atlas`.