So, You Want to Build a Beowulf?

Workload profiling and beowulf design (Part I of II)

Robert G. Brown
Duke University Physics Department

Abstract

Amdahl's Law is a relation that describes fundamental limits on how much (if any) benefit can be obtained by parallelizing any given application in the best of all possible worlds. However, the *real* world is even less kind and has a lot more variables. This introductory-level tutorial will focus on how to take a task, identify at least some of the parallelizable subtasks, determine the amount of time the task spends actually doing the parallelizable computations, determine the amount of time the parallel tasks will spend communicating for various node configurations between subtask epochs, and *then* pick a suitable beowulf architecture (if one exists, as there is no guarantee that one will).

The cost-benefit of various design alternatives (for example, clusters versus "true beowulfs" will also be discussed, as the economics of COTS cluster computing is at the heart of the ExtremeLinux effort.

Some time will be spent describing beowulf and cluster computing in broad terms, and with luck some of the points will be demonstrated with real code on a Real Live (if small) Beowulf!

Introduction
"The Standard Recipe"

1. Buy a pile of M^2COTS (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.
2. Add a nice, cheap 100BT NIC to each. Connect each NIC to nice, cheap 100BT switch to interconnect all nodes.
3. Add Linux and various "ExtremeLinux/Beowulf" packages to support distributed parallel computing; PVM, MPI, maybe more.
4. Blow your code away by running it in parallel...

...NOT!

Or perhaps more correctly, MAYBE. But wouldn't one like to know that one's code can be profitably run on a Beowulf BEFORE building it? And again, aren't there a few wee details glossed over in this "recipe"?

Steps in Building a Beowulf

Here is the One True Secret to building a successful beowulf. This has been certified over and over again on the beowulf list by virtually every "expert" on the list (and a few bozo's, like myself:-). BEFORE putting together even a paper plan for the beowulf:

- **Study your code.**
- **Study your code some more.**
- **Study it *still some more*.**
- ***Then* design and build your Beowulf or Cluster.**

The word "Study" here means to, if at all possible, use *measurements* and *prototyping* more than back-of-the-envelope estimates. Measurements are far more valuable than any theoretical estimate, however well-informed.

Profile your serial code and try running it on different candidate architectures. Try small parallelizations on just a few handy hosts connected by ANY network. Borrow a friend's beowulf for a few days if necessary.

Below I show in some detail both why and how to go about this...

First, some (very important) theory!

Amdahl's Law

The "speed" of a program is:

$$\frac{\text{Work}}{T} = \frac{\text{Work}}{T_s + T_p}$$

where T is the total execution time split up into T_s (the time spent doing things that have to be done serially - one after another) and T_p (the time spent doing things that *might* be doable in parallel).

Even with "perfect" parallelization and many (P) processors, the program cannot take less than T_s to complete. This 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.**

The result is expressed in fractions to get the RELATIVE speedup for a given amount of work. No matter how many processors are employed, if a calculation has a 10% serial component, the maximum speedup obtainable is 10.

Of course, reality is generally *even worse* than predicted by Amdahl's Law as we'll see later...

Profiling
The First Step

- **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. Is the time T_p spent in parallelizable subtasks 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$_{\text{tm}}$ 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.

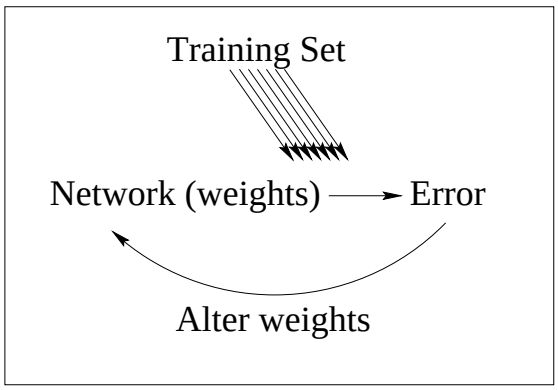

Figure 1: Training cycle of feed-forward, backprop network

- **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.**

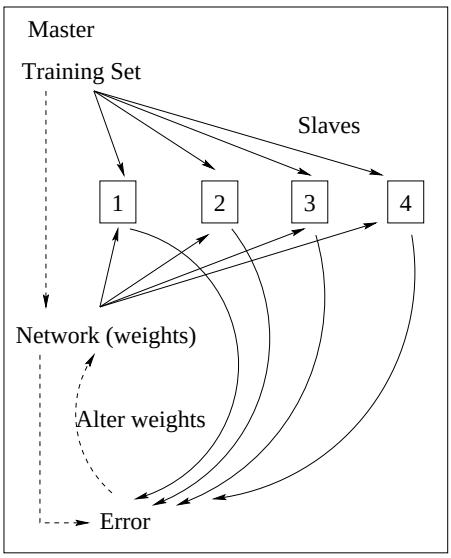

Figure 2: Training cycle of parallelized feed-forward, backprop network

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:

$$\frac{\text{Work}}{(\text{T}_\text{s}+\text{T}_\text{i,p})+\text{P}*\text{T}_\text{i,s}+\text{T}_\text{p}/\text{P}}$$

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.

OK, so T_i is "bad" and T_p is "good". Now what?

- T_i comes from the time required to communicate the information necessary to the node "during" (most often before and after) the execution of its parallel subtasks.
- In a Beowulf or cluster, this communication occurs over a commodity network. Faster networks therefore tend to shrink T_i, which is a Good Thing$_{\rm tm}$.
- One key beowulf feature is the ability (via kernel modifications) to combine several cheap COTS channels (like 100BT NIC's) into a single "channel bonded" communications link.
- Gigabit (per second) networks (Myrinet or Gigabit ethernet) that are rapidly becoming COTS products with the associated drop in price. Note that a gigabit is 125 Megabytes, which is very close to the speed of an underlying (33 MHz $\times$ 32 bit) PCI bus.

- **Nodes typically work from private memory. Private memory parallelization requires *message passing IPC's* to communicate data from memory on one node into memory on another.**
- **This is not the only possibility. Various sorts of *shared memory* constructs can be used as an IPC channel. In an SMP machine CPUs share real memory on a common bus. There are also CC-NUMA (Cache Coherent Non-Uniform Memory Access) models (and machines).**
- **Both MPI and PVM are basically message passing libraries and are designed for use on a network. MPI is good if you want your code to be easily portable to big iron. PVM is good if you plan to run on a heterogeneous network.**
- **Best possible network-based performance is obtained by using e.g. raw sockets and working incredibly hard. Not for the novice!**
- **Rule of thumb: Carefully explore the networking alternatives and match them to the problem. This technology is *very* rapidly evolving.**

We have to understand more about how the various times above interact in order to be able to pick an optimal design and implementation strategy. 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, subtasks may be:

- **Asynchronous, where they can proceed in a ragged fashion without affecting their mutual progress.**
- **Synchronous, where all subtasks must complete an epoch and exchange information to proceed.**

There are thus four "corners" to subtask space from coarse grained, asynchronous to fine grained synchronous (and a lot of ground in between).

Discussion

- **Granularity is very scale (problem or system size) dependent. One typically shifts granularity dramatically when changing the number of processors for fixed problem size or problem size for a fixed number of processors, both of which alter the T_p/P ratio *and* T_i.**
- **Granularity also obviously depends on communications speed and latency as indicated above.**
- **Coarse grained, asynchronous tasks are easy to program on nearly any beowulf or cluster design. There are many valuable tasks in this category, which partly explains the popularity of beowulfs and clusters.**
- **Coarse grained synchronous code can also run well on clusters, although it is harder to program and load balance efficiently.**
- **In particular, synchronous code are rate dominated by the *slowest* completion time, making load balancing key and heterogeneity bad. "Beowulfs" (homogeneous, isolated) are preferred to "clusters" for synchronous tasks.**
- **If parallelization at some point eliminates a key bottleneck or has an unusual scaling behavior because of the nature of the problem itself, a program can exhibit *super-linear speedup*.**

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 can be considered 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.

Beowulf hardware and software engineering is designed to MAKE YOUR PROBLEM *RELATIVELY* COARSE GRAINED ON THE (COTS) HARDWARE IN QUESTION and hence to keep $T_{i,s}$ 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.

Kids! Don't Try **This** at Home!

Parallel efficiency depends as much on *software design* and *algorithm* as it does on hardware, however much we focus on the latter because it is, frankly, easier to understand. Algorithms can scale *well* or *poorly*.

There is far more to be said here than I can communicate in a single, introductory tutorial. This is real computer science (and I'm not really a computer scientist) and the mathematics gets considerably more involved. I've probably done too much of the latter already!

Fortunately, I don't need to. Instead, I can refer you to an authoritative text like Almasi and Gottlieb. Suffice it to say that there is tremendous detail there and elsewhere and that algorithms (and their scaling relations) exist for many of the "standard" components one might expect to parallelize in a give program. Summing, sorting, FFT, Gauss elimination – all have good (and bad!) algorithms with known scaling rules.

The rule here is to not reinvent wheels (probably badly). Do some homework. To quote my net-friend Rob Ross of Clemson:

"You might want to discuss some of the paradigms for parallel programming: SPMD, master-slave, pipelining, etc. Picking the right approach for parallelizing can make all the difference in the world in the performance."

Consider it done...sort of.

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 (in light of the discussion above) we should have expected it.

Think and Improve

We made two changes in the code. First, we #ifdef-wrapped some debugging cruft in the slave code that was basically wasting time. (Debugging slaves is a problem in and of itself; the code was there for good reason.) Second, originally we multicast the network but sent each host its slice boundaries serially. This, in retrospect, was stupid, as the communication is 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 by Scaling

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 very distinct (if modest) speedup, even for this SMALL toy problem. We can guess that as we increase the size of the problem our parallelization speedup will improve with the granularity.

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 4 million or so 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$_{\mathrm{tm}}$ 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 do "office stuff" like read mail, write documents, 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 choosing the right 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 Beowulf or cluster 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.**
- **If your problem has components that have been properly parallelized by computer scientists somewhere, find the algorithm and use it. Don't be naive in your programming approach.**

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 sic gloria transit Microsoft.

References

First, the beowulf and linux-smp lists have to be the number one resource. Excellent people. Superb learning environment for years now. Check the archives – lots of great discussions past.

Second, "Highly Parallel Computing", by Almasi and Gottlieb. Very useful text for when you are ready to get serious.

Third, sooner or later your experiences will take you into "the literature". Finding "the best" algorithm is very likely to be worth the effort.