# Maximizing Beowulf Performance

Robert G. Brown
*Duke University Physics Department*
*Box 90305, Durham, NC, 27708-0305*
rgb@phy.duke.edu, http://www.phy.duke.edu/~rgb

## Abstract

At this point in time the beowulf (and other related compute cluster) architectures has come of age in Linux. Few indeed are those in any realm of technical computing that are unaware of the fact that one can assemble a collection of commodity off the shelf (COTS) computers and networking hardware into a high performance supercomputing environment. However, a detailed knowledge or appreciation for the bottlenecks and special problems associated with beowulf design is not so common. A review of the important bottlenecks and design features of a beowulf is given along with associated benchmarking and measurement tools to illustrate how to bridge the gap between the simple "recipe" of a beowulf as a pile of compute nodes, interconnected with a fast network and running linux and the realities of engineering a parallel code and beowulf-style cluster to achieve satisfactory performance.

## 1 Introduction

A very simple recipe for a beowulf [beowulf] might be: Purchase $N$ more or less identical COTS computers for compute nodes. Connect them by a COTS fast private network, for example switched fast ethernet. Serve them, isolate them and access them from auxiliary nodes (which may well be a single common "head node"). Install Linux and a small set of more or less standard parallel computation tools (typically PVM and MPI, along with several others that might or might not be present in a given installation). Voila! A beowulf supercomputer[1] Although this recipe is fair enough and will yield acceptable performance (measured in terms of "speedup" scaling as illustrated below) in many parallelized applications, in others it will not. In many cases the performance obtained will depend on the *details* of the node and network design as well as the design and implementation of the parallel application itself. The critical decisions made during the design process are informed by a deep and quantitative analysis of the fundamental rates and performance features of both the nodes and the network. The understanding of the important design criteria and how they correspond to features of the parallel application is a hallmark of a well-conceived beowulf project or proposal.

In the following sections we will briefly review the essential elements of successful beowulf design. They are:

- Understand Amdahl's Law and the fundamental limitations of parallel program design. The most common mistake of novices is to expect more from a parallelization of a program than is physically possible.
- Understand the fundamental rates (and latencies and bandwidths) of your computational hardware. In particular, know something about the fundamental latencies and bandwidths available in the various relevant subsystems (CPU, memory, network) and how they can act as bottlenecks to your parallelized subtasks. These rates should be known *quantitatively* to inform a sound beowulf design. Tools to use to make these measurements will be explicitly illustrated in the context of two simple beowulf nodes.

The goal of the discussion will be to convey an appreciation for some of the important design decisions to be made that is both *qualitative* and *quanti-*

[1] Much of this paper applies as well to any linux based cluster, including simple LAN clusters of workstations used as nodes. Many of these designs do not form, strictly speaking, a "beowulf supercomputer", but the term "beowulf" in this paper will be used for all beowulf-style clusters.

*tative*. The use of some mathematics is unavoidable but will be explained in the simplest possible terms.

With a general understanding of the scaling of parallel computation clusters of the general beowulf design in hand, it should be straightforward to select a successful and cost effective design for any parallelizeable problem that lies within the general range of the beowulf concept. Although it isn't a strict rule, the beowulf designs we will focus on are those appropriate for solving complex mathematical or numerical problems such as those encountered in physics, statistics, weather prediction, chemistry, and many other numerical venues.

This paper will *not* address clustering for purposes of failover and reliability or load balancing in e.g. a database server or webserver context. Although linux clusters are increasingly in use in these contexts, these clusters are not beowulfs.

It will also not address the more esoteric aspects of parallel program design (not intending at all to minimize the importance of a sound program design in successful beowulf operation) and indeed the mathematical treatment presented of parallel scaling may appear naive to those familiar with the entire multidimensional theory of the various algorithms that might be used to treat a given problem. We will instead be satisfied with providing references to some of the many authoritative works that address this subject far better than we would find possible in a short paper.

In the paper below, we will begin by addressing the basic theory of speedup and scaling in parallel computation. From this we will move on to a description of the important microscopic rates and measures that determine beowulf design and performance and a discussion of tools that can be used to measure them.

## 2 Amdahl's Law & Parallel Speedup

The theory of doing computational work in parallel has some fundamental laws that place limits on the benefits one can derive from parallelizing a computation (or really, any kind of work). To understand these laws, we have to first define the objective. In general, the goal in large scale computation is to get as much work done as possible in the shortest possible time within our budget. We "win" when we can do a big job in less time or a bigger job in the same time and not go broke doing so. The "power" of a computational system might thus be usefully defined to be the amount of computational work that can be done divided by the time it takes to do it, and we generally wish to optimize power per unit cost, or cost-benefit.

Physics and economics conspire to limit the raw power of individual single processor systems available to do any particular piece of work even when the dollar budget is effectively unlimited. The cost-benefit scaling of increasingly powerful single processor systems is generally nonlinear and very poor – one that is twice as fast might cost four times as much, yielding only half the cost-benefit, per dollar, of a cheaper but slower system. One way to increase the power of a computational system (for problems of the appropriate sort) past the economically feasible single processor limit is to apply more than one computational engine to the problem.

This is the motivation for beowulf design and construction; in many cases a beowulf may provide access to computational power that is available in a alternative single or multiple processor designs, but only at a far greater cost.

In a perfect world, a computational job that is split up among $N$ processors would complete in $1/N$ time, leading to an $N$-fold increase in power. However, any given piece of parallelized work to be done will contain parts of the work that *must* be done serially, one task after another, by a single processor. This part does *not* run any faster on a parallel collection of processors (and might even run more slowly). Only the part that can be parallelized runs as much as $N$-fold faster.

The "speedup" of a parallel program is defined to be the ratio of the rate at which work is done (the power) when a job is run on $N$ processors to the rate at which it is done by just one. To simplify the discussion, we will now consider the "computational work" to be accomplished to be an arbitrary task (generally speaking, the particular problem of greatest interest to the reader). We can then define the speedup (increase in power as a function of $N$) in terms of the time required to complete this particular fixed piece of work on 1 to $N$ processors.

Let $T(N)$ be the time required to complete the task on $N$ processors. The speedup $S(N)$ is the ratio

$$S(N) = \frac{T(1)}{T(N)}. \qquad (1)$$

In many cases the time $T(1)$ has, as noted above, both a serial portion $T_s$ and a parallelizeable portion $T_p$. The serial time does not diminish when the parallel part is split up. If one is "optimally" fortunate, the parallel time is decreased by a factor of $1/N$). The speedup one can expect is thus

$$S(N) = \frac{T(1)}{T(N)} = \frac{T_s + T_p}{T_s + T_p/N}. \qquad (2)$$

This elegant expression is known as *Amdahl's Law* [Amdahl] and is usually expressed as an inequality. This is in almost all cases the *best* speedup one can achieve by doing work in parallel, so the real speed up $S(N)$ is less than or equal to this quantity.

Amdahl's Law immediately eliminates many, many tasks from consideration for parallelization. If the serial fraction of the code is not much smaller than the part that could be parallelized (if we rewrote it and were fortunate in being able to split it up among nodes to complete in less time than it otherwise would), we simply won't see much speedup no matter how many nodes or how fast our communications. Even so, Amdahl's law is still far too optimistic. It ignores the overhead incurred due to parallelizing the code. We must generalize it.

A fairer (and more detailed) description of parallel speedup includes at least two more times of interest:

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

$\mathbf{T_{is}}$ The (average) additional *serial* time spent doing things like interprocessor communications (IPCs), setup, and so forth in all parallelized tasks. This time can depend on $N$ in a variety of ways, but the simplest assumption is that each system has to expend this much time, one after the other, so that the total additional serial time is for example $N * T_{is}$.

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

It is worth remarking that generally, the most important element that contributes to $T_{is}$ is the time required for communication between the parallel subtasks. This communication time is always there – even in the simplest parallel models where identical jobs are farmed out and run in parallel on a cluster of networked computers, the remote jobs must be begun and controlled with messages passed over the network. In more complex jobs, partial results developed on each CPU may have to be sent to all other CPUs in the beowulf for the calculation to proceed, which can be *very* costly in scaled time. As we'll see below, $T_{is}$ in particular plays an extremely important role in determining the speedup scaling of a given calculation. For this (excellent!) reason many beowulf designers and programmers are obsessed with communications hardware and algorithms.

It is common to combine $T_{ip}$, $N$ and $T_{is}$ into a single expression $T_o(N)$ (the "overhead time") which includes any complicated $N$-scaling of the IPC, setup, idle, and other times associated with the overhead of running the calculation in parallel, as well as the scaling of these quantities with respect to the "size" of the task being accomplished. The description above (which we retain as it illustrates the generic form of the relevant scalings) is still a *simplified* description of the times – real life parallel tasks can be much more complicated, although in many cases the description above is adequate.

Using these definitions and doing a bit of algebra, it is easy to show that an improved (but still simple) estimate for the parallel speedup resulting from splitting a particular job up between $N$ nodes (assuming one processor per node) is:

$$S(N) = \frac{T_s + T_p}{T_s + N * T_{is} + T_p/N + T_{ip}}. \qquad (3)$$

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.

It is useful to plot the dimensionless "real-world speedup" (3) for various *relative* values of the times. In all the figures below, $T_s = 10$ (which sets our basic scale, if you like) and $T_p = 10$, 100, 1000, 10000, 100000 (to show the systematic effects of parallelizing more and more work compared to $T_s$).

The primary determinant of beowulf scaling performance is the amount of (serial) work that must be done to set up jobs on the nodes and then in com-

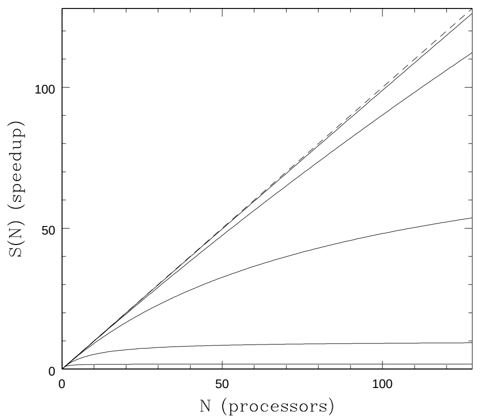


Figure 1: $T_{is} = 0$ and $T_p = 10$, 100, 1000, 10000, 100000 (in increasing order).

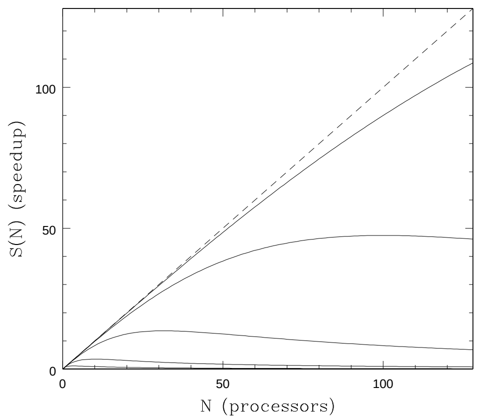


Figure 2: $T_{is} = 10$ and $T_p = 10$, 100, 1000, 10000, 100000 (in increasing order).

munications between the nodes, the time that is represented as $T_{is}$. All figures have $T_{ip} = 1$ fixed; this parameter is rather boring as it effectively adds to $T_s$ and is often very small.

Figure 1 shows the kind of scaling one sees when communication times are negligible compared to computation. This set of curves is roughly what one expects from Amdahl's Law alone, which was derived with no consideration of IPC overhead. Note that the dashed line in all figures is perfectly linear speedup, which is never obtained over the entire range of $N$ although one can often come close for small $N$ or large enough $T_p$.

In figure 2, we show a fairly typical curve for a "real" beowulf, with a relatively small IPC overhead of $T_{is} = 1$. In this figure one can see the advantage of cranking up the parallel fraction ($T_p$ relative to $T_s$) and can also see how even a relatively small serial communications process on each node 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. Increasing $T_{is}$ further (relative to everything else) causes the speedup curves to peak earlier and at smaller values.

Finally, in figure 3 we continue to set $T_{is} = 1$, but this time with a *quadratic* $N$ dependence $N^2 * T_{is}$ of the serial IPC time. This might result if the communications required between processors is long range (so every processor must speak to every other processor) and is not efficiently managed by a suitable algorithm. There are other ways to get nonlinear dependences of the additional serial time on $N$, and as this figure clearly shows they can have a profound effect on the per-processor scaling of the speedup.

As one can clearly see, unless the ratio of $T_p$ to $T_{is}$ is in the ballpark of 100,000 to 1 one cannot actually *benefit* from having 128 processors in a "typical" beowulf. At only 10,000 to 1, the speedup saturates at around 100 processors and then decreases. When the ratio is even smaller, the speedup peaks with only a handful of nodes working on the problem. From this we learn some important lessons. The most important one is that for many problems simply adding processors to a beowulf design won't provide any additional speedup and could even slow a calculation down *unless one also scales up the problem* (increasing the $T_p$ to $T_{is}$ ratio) as well.

The scaling of a given calculation has a significant impact on beowulf engineering. Because of overhead, speedup is not a matter of just adding the speed of however many nodes one applies to a given problem. For some problems it is clearly advantageous to trade off the *number* of nodes one purchases (for example in a problem with small $T_s$ and $T_p/T_{is} \approx 100$) in order to purchase tenfold improved communications (and perhaps alter the $T_p/T_{is}$ ratio to 1000).

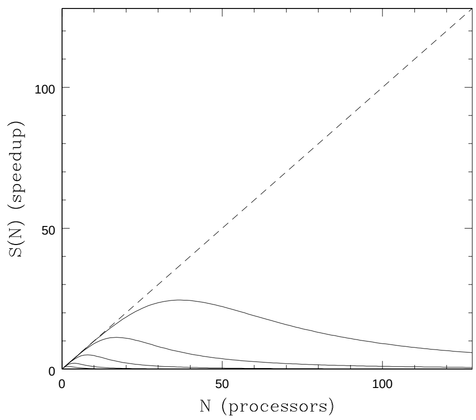


Figure 3: $T_{is} = 10$ and $T_p = 10$, 100, 1000, 10000, 100000 (in increasing order) with $T_{is}$ contributing *quadratically* in $N$.

The nonlinearities prevent one from developing any simple rules of thumb in beowulf design. There are times when one obtains the greatest benefit by selecting the fastest possible processors and network (which reduce both $T_s$ and $T_p$ in absolute terms) instead of buying more nodes because we know that the rate equation above will limit the parallel speedup we might ever hope to get even with the fastest nodes. Paradoxically, there are other times that we can do better (get better speedup scaling, at any rate) by buying *slower* processors (when we are network bound, for example), as this can also increase $T_p/T_{is}$. In general, one should be aware of the peaks that occur at the various scales and not naively distribute small calculations (with small $T_p/T_{is}$) over more processors than they can use.

In summary, parallel performance depends primarily on certain relatively simple parameters like $T_s$, $T_p$ and $T_{is}$ (although there may well be a devil in the details that we've passed over). These parameters, in turn are at least partially under our control in the form of programming decisions and hardware design decisions. Unfortunately, they depend on many microscopic measures of system and network performance that are inaccessible to many potential beowulf designers and users. $T_p$ clearly should depend on the "speed" of a node, but the single node speed itself may depend *nonlinearly* on the speed of the processor, the size and structure of the caches, the operating system, and more.

Because of the nonlinear complexity, there is no way to *a priori* estimate expected performance on the basis of any simple measure. There is still considerable benefit to be derived from having in hand a set of quantitative measures of "microscopic" system performance and gradually coming to understand how one's program depends on the potential bottlenecks they reveal. The remainder of this paper is dedicated to reviewing the results of applying a suite of microbenchmark tools to a pair of nodes to provide a quantitative basis for further beowulf hardware and software engineering.

## 3 Microbenchmarking Tools

From the previous section we can see that there are several things that we have to understand fairly thoroughly to design a beowulf to cost-effectively tackle a given problem. To achieve the best scaling behavior, 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 tradeoffs 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.

Finding the truly optimum design can be somewhat 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 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. In all cases, the design process is significantly easier if one possesses a detailed and quantitative knowledge of various microscopic *rates, latencies,* and *bandwidths.*

- 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[2] but 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)

[2] More time was spent by the author of this paper working on and with the tool than on the paper:-)

[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.

### 3.1 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 be-

| | |
|---|---|
| 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 1: Lucifer System Description

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

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

Table 3: Lmbench latencies for context switches, in microseconds (smaller is better).

| | |
|---|---|
| pipe | $10.62 \pm 0.069$ |
| AF UNIX | $33.74 \pm 3.398$ |
| UDP | $55.13 \pm 3.080$ |
| TCP | $127.71 \pm 5.428$ |
| TCP Connect | $265.44 \pm 7.372$ |
| RPC/UDP | $140.06 \pm 7.220$ |
| RPC/TCP | $185.30 \pm 7.936$ |

Table 4: Lmbench *local* communication latencies, in microseconds (smaller is better).

| | |
|---|---|
| UDP | $164.91 \pm 2.787$ |
| TCP | $187.92 \pm 9.357$ |
| TCP Connect | $312.19 \pm 3.587$ |
| RPC/UDP | $210.65 \pm 3.021$ |
| RPC/TCP | $257.44 \pm 4.828$ |

Table 5: Lmbench *network* communication latencies, in microseconds (smaller is better).

| | |
|---|---|
| L1 Cache | $6.00 \pm 0.000$ |
| L2 Cache | $112.40 \pm 7.618$ |
| Main mem | $187.10 \pm 1.312$ |

Table 6: Lmbench *memory* latencies in nanoseconds (smaller is better). Also see graphs for more complete picture.

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

Table 7: Lmbench *local* communication bandwidths, in $10^6$ bytes/second (bigger is better).

| | |
|---|---|
| TCP | $11.21 \pm 0.018$ |
| UDP | (not available) |

Table 8: Lmbench *network* communication bandwidths, in $10^6$ bytes/second (bigger is better).

| | |
|---|---|
| Single precision | $289.10 \pm 1.394$ |
| Double precision | $299.09 \pm 2.295$ |

Table 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.

owulf 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.

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.

### 3.2 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

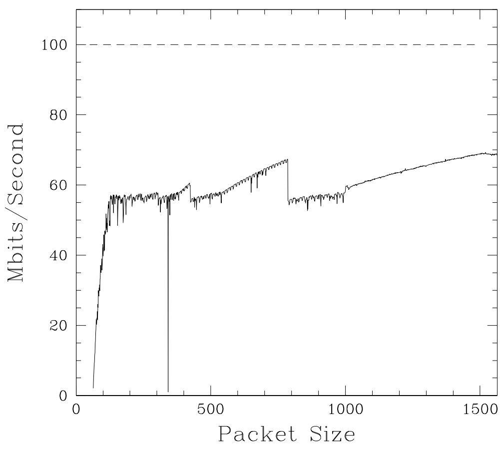


Figure 4: TCP Stream (netperf) measurements of bandwidth as a function of packet size between lucifer and eve.

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 lucifer and "eve" (a reasonably similar host on the same switch) as a function of packet size. These results are shown in figure 4.

Figure 4 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.

### 3.3 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

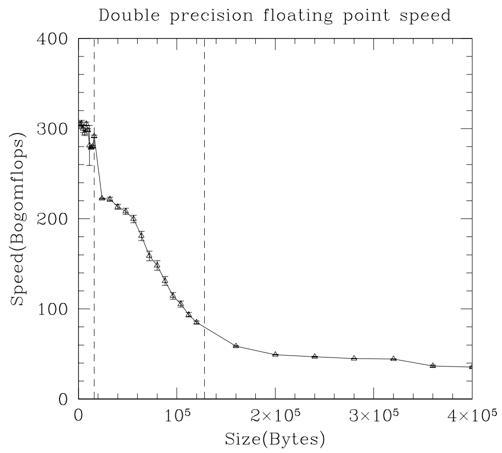


Figure 5: Double precision floating point operations per second as a function of vector length (in bytes). All points average 100 independent runs. The dashed lines indicate the locations of the L1 and L2 cache boundaries.

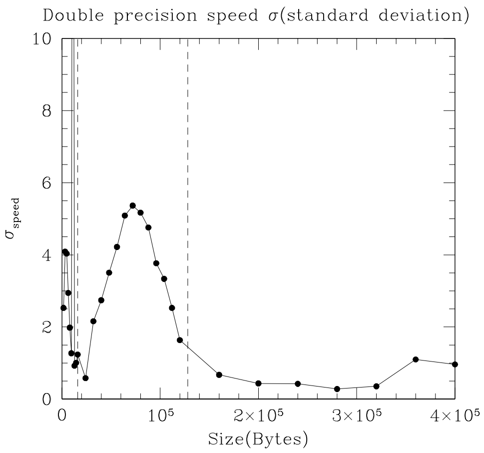


Figure 6: The standard deviation (error) associated with figure 5.

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 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 5.

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 6 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.

## 4 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.

## 5 Acknowledgments

I would like to gratefully acknowledge the support of Duke University, the Army Research Office, Intel Corporation, who variously funded the author and/or one of his beowulfs. I am very grateful to this paper's "shepherd", Walter B. Ligon III, for reading the various drafts and making useful suggestions, and to both Larry McVoy and Carl Staelin for at least listening to my passionate plea for an unrestricted GPL for the lmbench suite and for adding a number of beowulf friendly measures. Finally, I would also especially like to thank the members and regular participants on the beowulf list.

## 6 Availability

All software discussed in this paper is open source and readily available over the network at the URL's indicated in the bibliography below or by sending email to the author at rgb@phy.duke.edu. The particular kind of licensing for each (GPL or not) is indicated in the reference.

## References

[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`.