# Maximizing Beowulf Performance

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

**This work was supported by the**
**U. S. Army Research Office**

**10/14/00**

# "Recipe" for a Beowulf

- **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 (PVM, MPI,...).**

**Often works, sometimes doesn't. The difference is in the details.**

**Critical design decisions (of both a parallel program and a beowulf to run it on) are informed by a deep and quantitative understanding of the *fundamental rates* of the *nodes* and the *network* and how they dictate program performance and scaling.**

**In previous presentation[1] the *programming* of parallel applications was considered. In this presentation, we will focus on the hardware.**

[1] http://www.phy.duke.edu/brahma/beowulf_advanced.ps

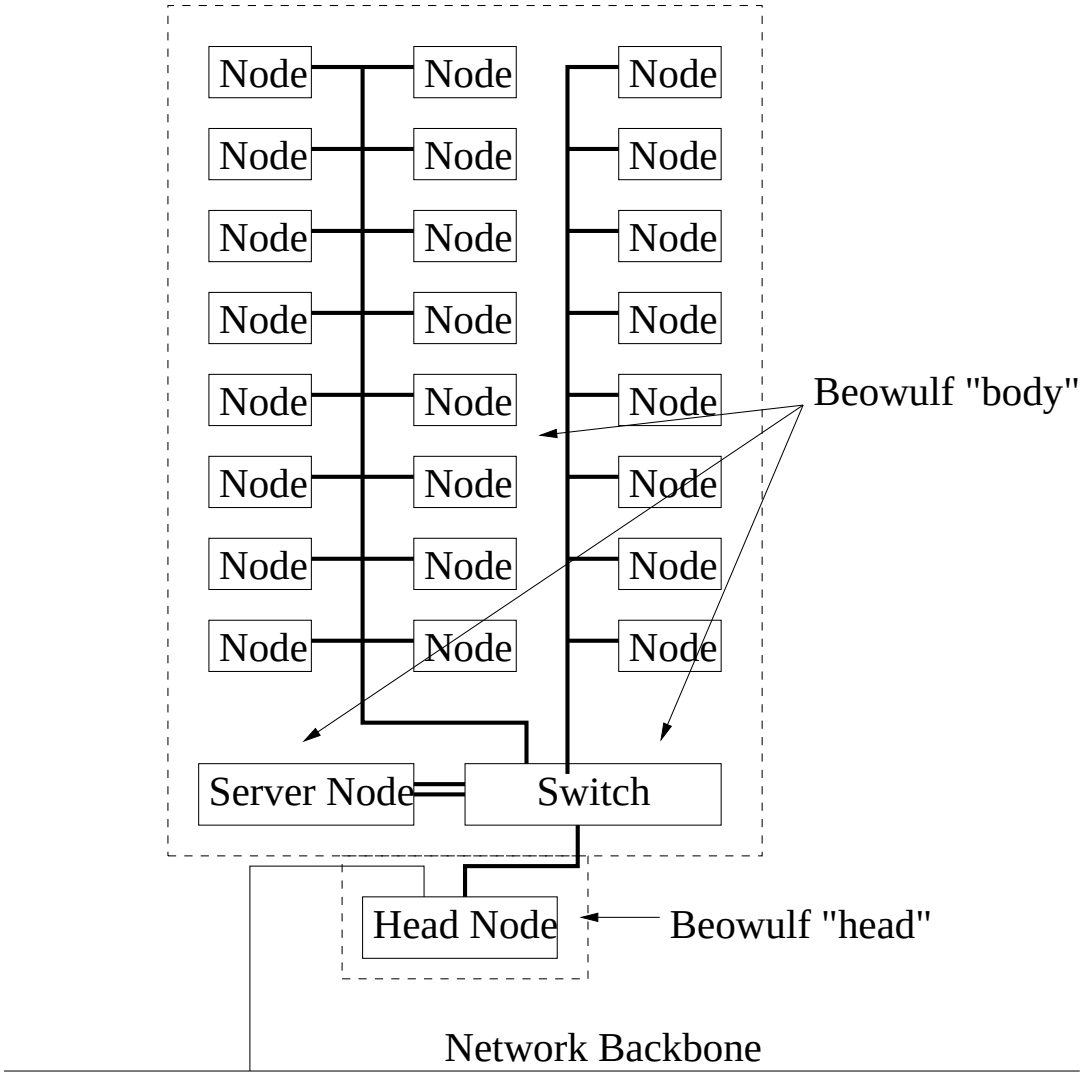


Figure 1: **Typical "Recipe" Beowulf.**

# Amdahl's Law

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

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

**$T(1) =$ "serial time" $T_s$ + "parallel(izeable) time" $T_p$.**

**The *best* speedup one can expect is thus:**

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

**This is *Amdahl's Law* (usually expressed as an inequality)**

**Amdahl's Law immediately eliminates many, many tasks from consideration for parallelization. However, Amdahl's law is still far too optimistic (overhead incurred due to parallelizing the code). We must generalize it.**

**$T_s$** **The original single-processor serial time.**

**$T_{is}$** **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 $N * T_{is}$.**

**$T_p$** **The original single-processor parallelizeable time.**

**$T_{ip}$** **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.**

**Improved estimate of speedup on $N$ nodes is:**

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

**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.**

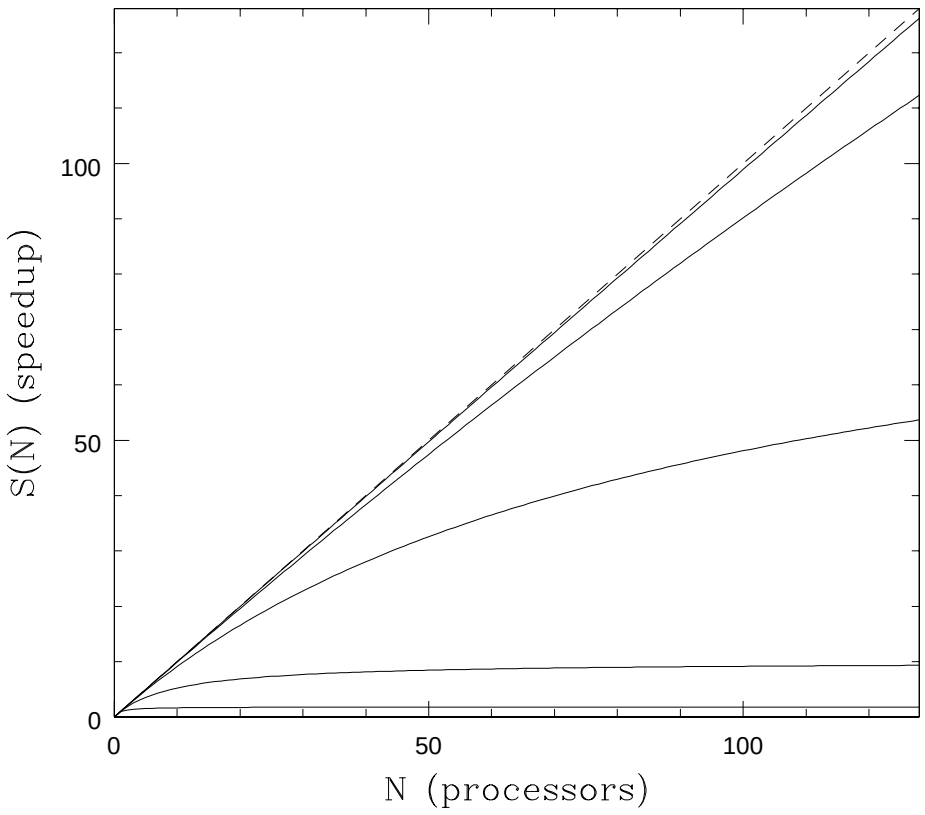


Figure 2: $T_{is} = 0$ **and** $T_p$ **= 10, 100, 1000, 10000, 100000 (in increasing order).**

**It is useful to plot the dimensionless "real-world speedup" (3) for various *relative* values of the times. Let** $T_s$ **= 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** $T_s$**).** $T_{ip} = 1$ **fixed (often very small).**

**Figure 1 shows just about pure Amdahl's Law scaling for various parallel fractions.**

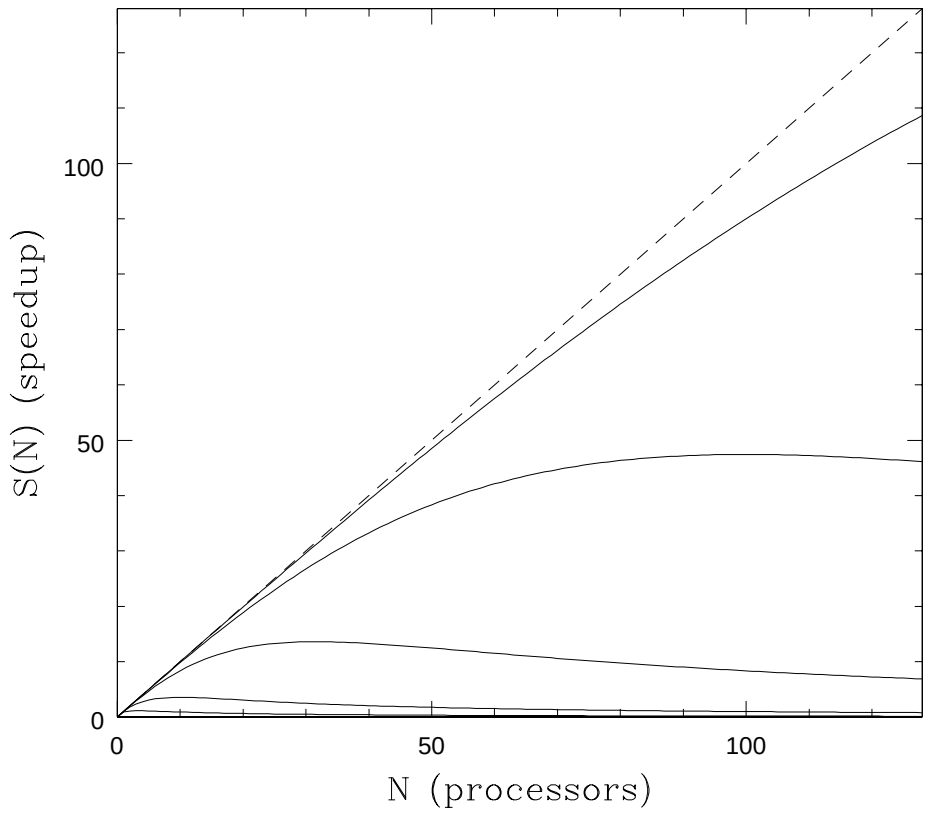


Figure 3: $T_{is} = 10$ **and** $T_p =$ **10, 100, 1000, 10000, 100000 (in increasing order).**

**Fairly typical curve for a "real" beowulf, with a relatively small IPC overhead of** $T_{is} = 1$**.** $T_p \gg T_s$ **"good". Even 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.**

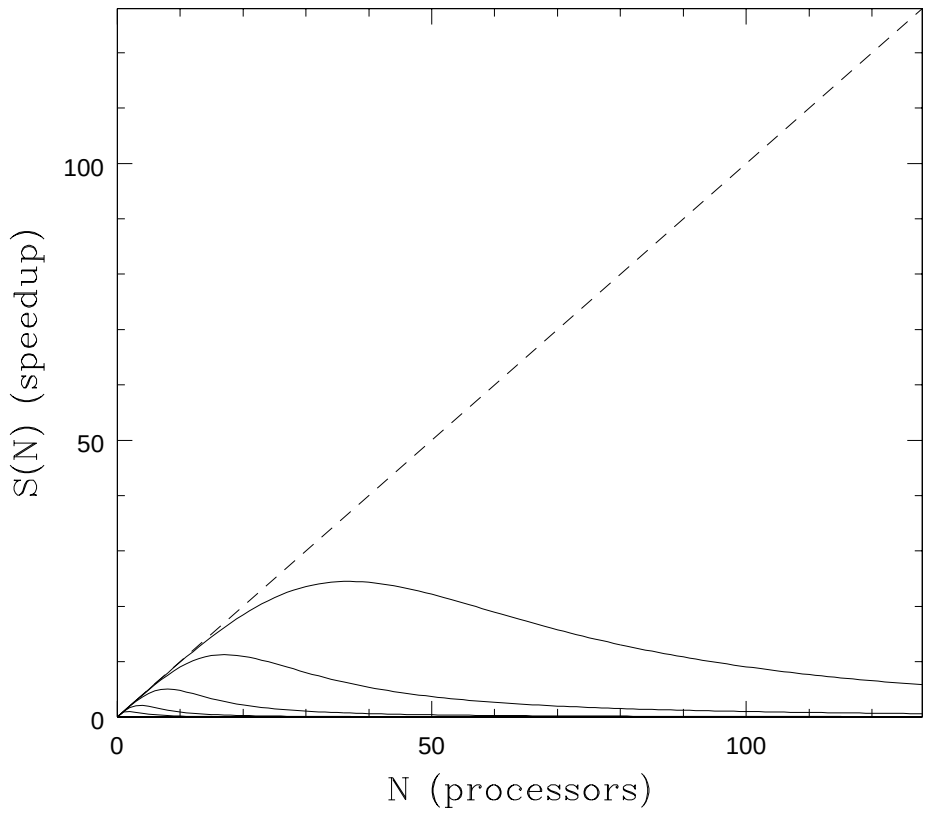


Figure 4: $T_{is} = 10$ **and** $T_p =$ **10, 100, 1000, 10000, 100000 (in increasing order) with** $T_{is}$ **contributing *quadratically* in** $N$**.**

**In figure 3** $T_{is} = 1$**, but with *quadratic*** $N$ **dependence** $N^2 * T_{is}$ **of the serial IPC time. (Can happen due to communications topology, long range communications). Can have *nonlinear* dependences of the additional serial time on** $N$ **with a profound effect on the per-processor scaling of the speedup.**

# Lessons

- **Bigger beowulf is not always better beowulf.**
- **Faster nodes are not always better nodes.**
- **Many nonlinearities and tradeoffs in *both* code design *and* hardware design.**
- **Our "beowulf recipe" has a moderate capacity for failure if naively applied.**

## The Best Plan

- ***Study* your problem (with Real Parallel Programming book in hand).**
- ***Learn* something about the interactions and dependencies of performance on node hardware and networking design.**
- ***Combine* the two with benchmarking and prototyping.**
- ***Recognize* that beowulfery is all about *cost-benefit optimization*.**

**As a final note, accept that *some* problems may not yet run on a beowulf architecture, at least not one that you can afford.**

# Measuring Hardware Performance

## Microbenchmarks and Sweeps

**$T_s$, $T_p$ and $T_{is}$ depend on:**

- **Type, speed, cache structure of processor.**
- **Type, speed, structure of memory.**
- **Compiler and libraries used.**
- **OS Kernel.**
- **Network (with its many dimensions and choices).**
- **. . . and more.**

**Recognizing this, it is silly to try to characterize nonlinear, complex systems performance in terms of a single number, but a beowulf FAQ is certainly "how many GFLOPS will my $N$ node beowulf have?"**

**Still some purpose to benchmarks, but the only *reliable* macroscopic benchmark is *your code*.**

**Let us see how system design decisions can be better informed by *microscopic* benchmarks (that measure performance in only one fairly narrow task), ideally across a *sweep* of key system parameters.**

# Microbenchmarking Tools
## Lmbench, Netperf, CPU-Rate

**We typically wish to measure *rates*, *latencies* and *bandwidths* of particular system components with microbenchmarking tools.**

**Exploiting advantages of the nonlinearities (or avoiding their *dis*advantages) can result in dramatic improvements in performance: e.g. – ATLAS. It would be useful to have a set of basic system rates automatically built and maintained by a microbenchmark daemon and made available in e.g. /proc/rates.**

**Larry McVoy and Carl Staelin's "lmbench" toolset has the promise of becoming *the* fundamental toolset to support systems engineering and cluster design. It is still incomplete for this purpose (but improving rapidly). Supplemented below a privately written "cpu-rate" tool.**

**Applied to (among other systems) "lucifer": a 466 MHz *dual* Celeron system in my home beowulf. The cpu-rate results are also included on this page (lmbench had no microscopic CPU performance measuring tools when this project was begun).**

| | |
|---|---|
| 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**

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

Table 2: **Lmbench *memory* latencies in nanoseconds (smaller is better).**

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

Table 3: **CPU-rates in BOGOMFLOPS – $10^6$ simple arithmetic operations/second, in L1 cache (bigger is better). However, *see graph!* This single "number" hides more than it reveals!**

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

Table 4: **Lmbench *network* communication bandwidths, in $10^6$ bytes/second (bigger is better). Again, *see graph!***

| | |
|---|---|
| 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 5: Lmbench latencies for context switches, in microseconds (smaller is better).

| | |
|---|---|
| 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 6: 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".

**NOTE to anyone who can read this in the brief time I flash it onto the screen: The LMbench "rules" require that all the LMbenchmark results for a machine be published or presented at one time or none at all. So look, I've published it. At least to those with good eyes...;-)**

| | |
|---|---|
| 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 7: 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 8: Lmbench *network* communication latencies, in microseconds (smaller is better).

| | |
|---|---|
| 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 9: Lmbench *local* communication bandwidths, in $10^6$ bytes/second (bigger is better).

# Arggggh!
# What were all those Damn Numbers?

**lmbench clearly produces an *extremely detailed* picture of microscopic systems performance. Actually, examined closely, many of these numbers are of obvious interest to beowulf and system and kernel designers (yes, Linus Torvalds Himself uses lmbench).**

**However, we must focus in order to conduct a sane discussion. Let's look *only* at the:**

- **The network**
- **The memory**
- **The "cpu-rates"**

**Single numbers seem somehow too, what, zero dimensional? Let's at least study these things as functions in one dimension by *sweeping* over an important parametric range and plotting the results so we can *see* them.**

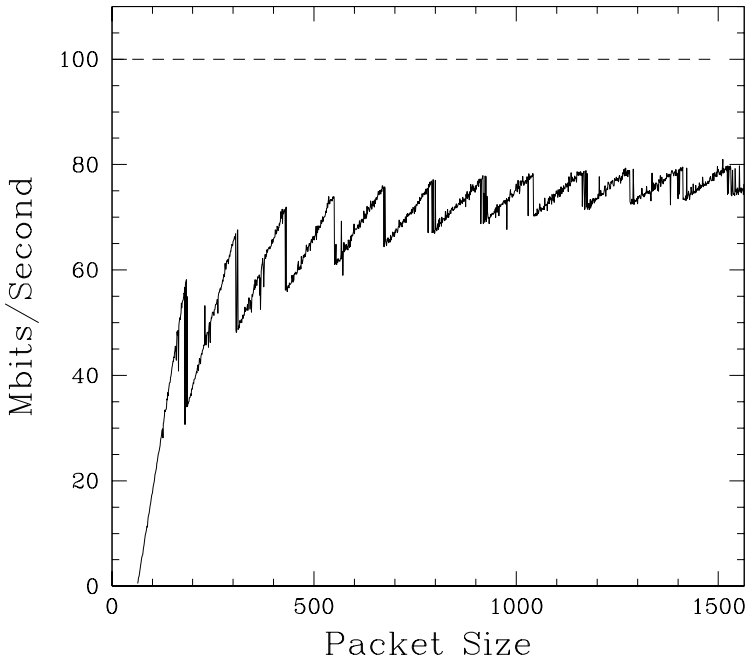


Figure 5: **TCP Stream measurements of bandwidth as a function of packet size between lucifer and eve.**

# bw_tcp Results

- **lmbench: 150-200 microseconds/one-byte TCP message (on lucifer) or at most 5000-7000 packets can be sent per second. For small packets the bandwidth observed is dominated by *latency*.**
- **It also reveals some interesting and unexpected structure. Bad [card|switch|driver|kernel|...]?**
- **Clearly my home beowulf needs work if it is to deliver top networking performance. It is not enough to know that a card/switch combination "works with linux".**

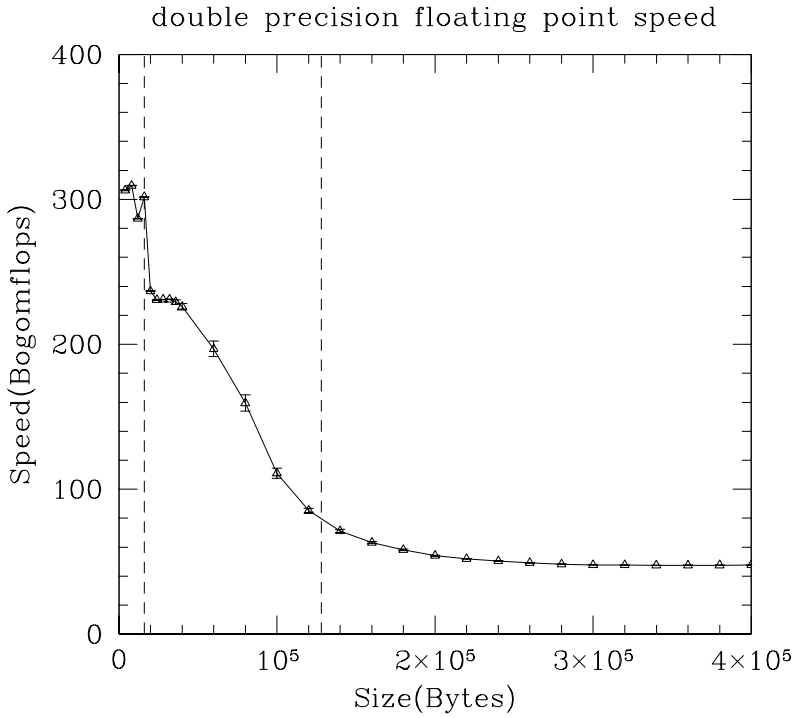


Figure 6: **Double precision floating point operations per second as a function of vector length (in bytes). All points average 100 independent runs. The dashed lines indicate the locations of the L1 and L2 cache boundaries.**

# CPU Results: cpu-rate

- `x[i] = (1.0 + x[i])*(1.5 - x[i])/x[i];` ($x[i] = 1.0$).
- **Still, MFLOPS are somewhat bogus – "bogomflops".**
- **Still, it's what I think "most people" mean when they ask how fast a system can do floating point arithmetic.**

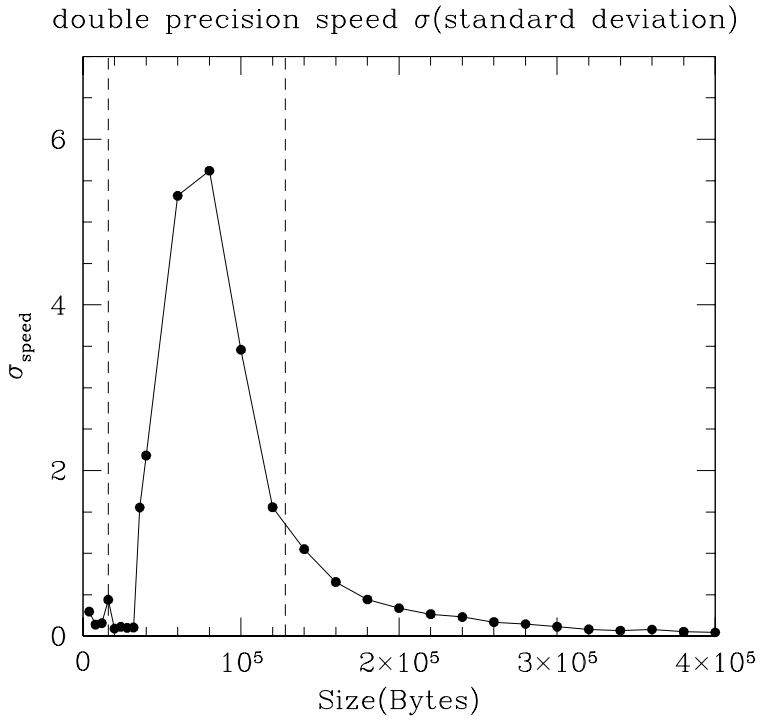


Figure 7: **The standard deviation (error) associated with figure 6.**

# Cool Fact: L2 Susceptibility Peak

- **L2 cache boundary (so I'm told) difficult to pick out of a benchmark (L1, from previous figure, is easy).**
- **In many independent runs, averaged, a clear peak in the variance across L2 region. L2 cache size power of 2, so easy to pick out.**
- **Measurable effect of (lack of) page coloring (which might reduce the fluctuations in the L2 latency)? Don't know for sure.**

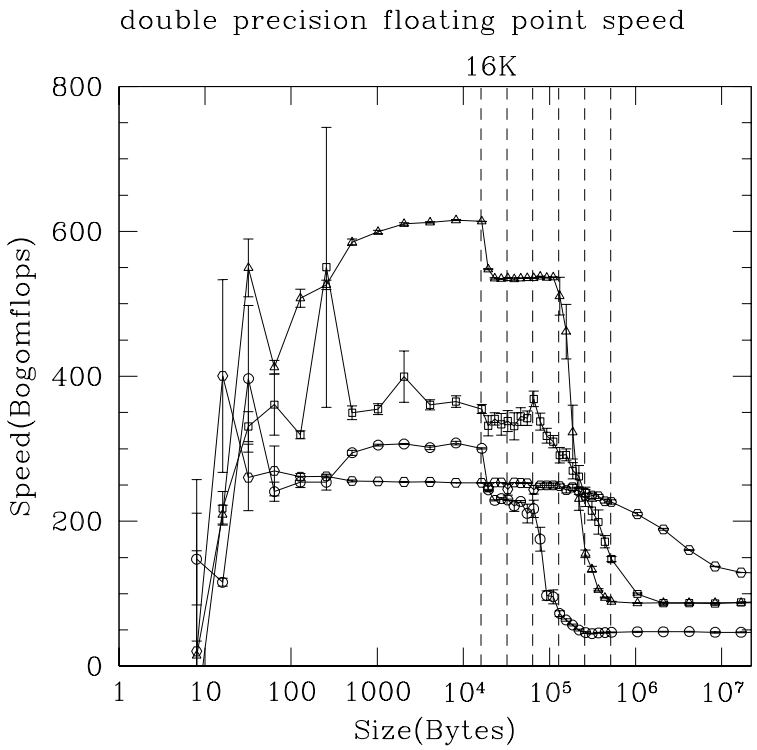


Figure 8: **A beautiful figure comparing four "important" potential beowulf nodes**

# Comparative Results

- **933 MHz PIII (triangles) *much* faster in L1 and out to first half of L2!**
- **667 MHz 2.264 Alpha (hexagons) weak in L1 (no better than a Celeron!) but ever so strong out of main memory.**
- **750 MHz Athlon (squares: older generation, not e.g. Duron) probable cost-benefit winner.**
- **466 MHz Celeron (circles) looks like it finally is dead – or is it?**

| Clock/CPU | Time (seconds) | Clock*Time |
|---|---|---|
| 200 MHz PPro | 97.35 | 19470 |
| 300 MHz PII | 68.46 | 20538 |
| 300 MHz Celeron | 66.06 | 19818 |
| 466 MHz Celeron | 46.48 | 21660 |
| 900 MHz Athlon | 28.7 | 25830 |
| 667 MHz Alpha EV67 | 23.79 | 15868 |
| 933 MHz PIII | 21.11 | 21111 |

Table 10: **OnSpin3d times *corrected for CPU clock!* (Smaller MHz*Time is better.)**

# Meditations upon the purpose of cache...

***My* favorite "benchmark" is the Monte Carlo program I use in my physics research. I've run it on many years of systems and CPUs back to Sparc 1 and 2's.**

- **Intel P6 family scales with ONLY clock from PPro - PIII, including the Celeron.**
- **Cache memory really works! For me, anyway.**
- **Exercise for the audience: Correct these results for the *cost* of the systems to pick a cost-benefit winner!**

# Conclusions

**With these microbenchmark 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 (better yet) simultaneously develop, improve, and tune a *matched* beowulf design and parallel program *together*.**
- **Collecting comparative results can enable one to do the all-important optimization of *cost-benefit* that is really the fundamental motivation for using a beowulf design in the first place.**
- **Finally, I have dreams of an automatically generated set of system tuning parameters that can be read by and used by userspace programs and libraries – e.g. a portable, truly "automatically tuned" ATLAS?**

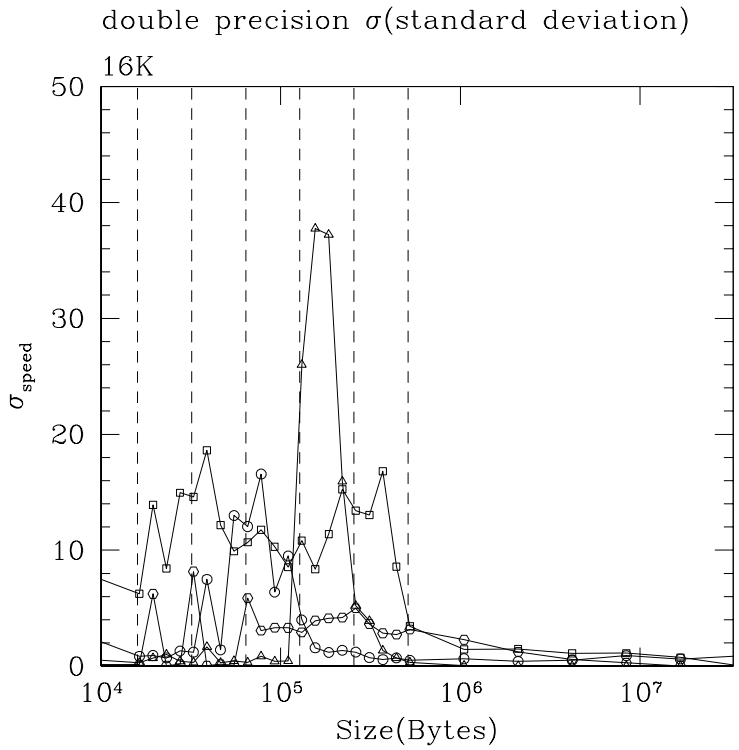


Figure 9: **A beautiful figure comparing the L2 *sigma* for four "important" potential beowulf nodes**

# Comparative Sigma Results

- **PIII (triangles) sharp peak across rapid drop off, ends at 256 KB L2 boundary.**
- **Alpha (hexagons) hardly any peak at all (4 MB L2, after all) but weak boundary at 4 MB..**
- **Athlon (squares) sharp peak at 512 KB L2 cache, but then "matches" Alpha.**
- **Celeron (circles) sharp peak ends at 128 KB L2**

| Function | Rate (MB/s) | RMS time | Min time | Max time |
|---|---|---|---|---|
| Copy | 136.9658 | 0.2340 | 0.2336 | 0.2348 |
| Scale | 154.7853 | 0.2070 | 0.2067 | 0.2086 |
| Add | 189.6199 | 0.2533 | 0.2531 | 0.2534 |
| Triad | 174.0612 | 0.2760 | 0.2758 | 0.2761 |

Table 11: **Stream Results for lucifer.**

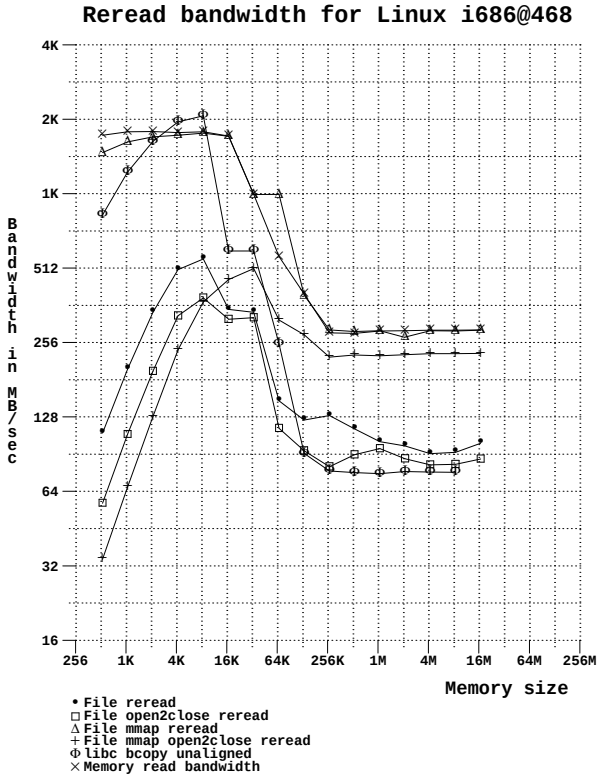


Figure 10: **LMbench memory read results plotted.**

# Memory Stuff

# 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`.