

Performance Measuring on Blue Horizon and Sun HPC Systems:

Timing, Profiling, and Reading
Assembly Language

NPACI Parallel Computing Institute 2000

Sean Peisert
peisert@sdsc.edu

Purpose

- Applications, as they are first written, can be initially very slow.
- Sometimes, even the most well-planned code can be made to run one or more orders of magnitude faster.
- To speed up applications, one must understand what is happening in the application.

Techniques

- By *timing* code, one can understand how fast or slow an application is running but not how fast it can potentially run.
- By *profiling* code, one can understand where the application is taking the most time.
- By reading *assembly language*, one can understand if the sections that the profiler identifies as slow are acceptable or poorly compiled.

Benefits

- By *tuning* code in a knowledgeable way, one can often significantly speed up an application.
- Using the techniques of *timing*, *profiling*, and reading *assembly language*, one can make educated guesses about what to do instead of shooting blindly.

Timing Terms

- Code for a single node:
 - Wallclock time
 - CPU time
- Code for a parallel machine:
 - Computation time
 - Communication time
 - Latency
 - Bandwidth

Timing on Parallel Machines

- Latency is the time it takes to send a message from one processor to another.
- Bandwidth is the amount of data in a given time period that can be sent from one processor to another.

$$[\text{communication time}] = [\text{startup time}] + [\text{message size}]/[\text{bandwidth}]$$

Timing Latency

- Different machines might be suited for coarse or fine-grained communication.
- The Sun HPC system and Blue Horizon both do fairly well intra-node, but inter-node communication is slower.
- Run 'ring' benchmarks to time communication latency.

Ring

- Pass messages from one processor to the next and back to the first in a ring fashion.
- Have it do **multiple cycles** (it has to warm up).
- Increase the size of the message passed until the time to pass it stabilizes.
- It will help to characterize the performance of message-passing to determine how large the messages in a “real” program can/should be.

Timing on Parallel Machines Tips

- Make sure that the system clocks on all machines are the same.
- In addition to the time for communication and computation, there is also “waiting.”
- Remember that some forms of communication (i.e. `MPI_Recv()`) are “blocking.”
- Goal is to minimize waiting and communication relative to computation.

Performance Measuring with Timings: Wallclock

- Wallclock time (real time, elapsed time)
 - High resolution (unit is typically 1 μ s)
 - Best to run on dedicated machines
 - Good for inner loops in programs or I/O.
 - First run may be varied due to acquiring page frames.

Performance Measuring with Timings: CPU

- CPU time
 - **User** Time: instructions, cache, & TLB misses
 - **System** time: initiating I/O & paging, exceptions, memory allocation
 - Low resolution (typically 1/100 second)
 - Good for whole programs or a shared system.

Timing Tips

- Wallclock time contains everything that CPU time contains but it also includes waiting for I/O, communication, and other jobs.
- For any timing results use several runs (three or more) and use the *minimum time*, not the average times.

Wallclock Time

- `gettimeofday()` — C/C++
 - Resolution up to microseconds.
- `MPI_Wtime()` — C/C++/Fortran
- Others: `ftime`, `rtc`, `gettimer`, ...

- Both Blue Horizon and “gaos” (Sun HPC) have `gettimeofday()`, `MPI_Wtime()`, and `ftime`.

gettimeofday()

C++ Example

```
#include <sys/time.h>
struct timeval *Tps, *Tpf;
void *Tzp;
Tps = (struct timeval*) malloc(sizeof(struct timeval));
Tpf = (struct timeval*) malloc(sizeof(struct timeval));
Tzp = 0;
gettimeofday (Tps, Tzp);
    <code to be timed>
gettimeofday (Tpf, Tzp);
printf("Total Time (usec): %ld\n",
      (Tpf->tv_sec-Tps->tv_sec)*1000000
      + Tpf->tv_usec-Tps->tv_usec);
```

MPI_Wtime() C++ Example

```
#include <mpi.h>
double start, finish;

start = MPI_Wtime();
    <code to be timed>
finish = MPI_Wtime();

printf("Final Time: %f", finish-start);
/* Time is in milliseconds since a particular date */
```

CPU Timing

- For timing the entire execution, use UNIX 'time'
 - Gives user, system and wallclock times.
- For timing segments of code:
- ANSI C

```
#include <times.h>
```

```
Clock_t is type of CPU times
```

```
clock() / CLOCKS_PER_SEC
```

CPU Timing

- `SYSTEM_CLOCK()` — Fortran (77, 90)
 - Resolution up to microseconds

SYSTEM_CLOCK()

```
INTEGER TICK, STARTTIME, STOPTIME, TIME
CALL SYSTEM_CLOCK(COUNT_RATE = TICK)
...
CALL SYSTEM_CLOCK (COUNT = STARTTIME)
  <code to be timed>
CALL SYSTEM_CLOCK (COUNT = STOPTIME)

TIME = REAL(STOPTIME-STARTTIME) / REAL(TICK)

PRINT 4, STARTTIME, STOPTIME, TICK
4      FORMAT (3I10)
```

Example `time` Output

```
5.250u 0.470s 0:06.36 89.9% 7787+30041k 0+0io 805pf+0w
```

- 1st column = user time
- 2nd column = system time
- 3rd column = total time
- 4th column = (user time + system time)/total time in %. In other words, the percentage of time your job gets alone.
- 5th column = (possibly) memory usage
- 7th column = page faults

time Tips

- Might need to specifically call `/usr/bin/time` instead of the built-in `time`.
- Look for low “system” time. A significant system time may indicate many exceptions or other abnormal behavior that should be corrected.

More About Timing

- Compute times in cycles/iteration and compare to plausible estimate based on the assembly instructions.
- $\frac{([\text{program time}] - [\text{initialization time}] * [\text{clock speed in Hz}])}{[\text{number of cycles}]}$

More About Timing

- Compute time of the program using only a single iteration to determine how many seconds of timing, loop, and execution overhead are present in every run.
- Subtract the overhead time from each run when computing cycles/iteration.
- Make sure that the system clock on each machine is the same time.

Profiling – Where does the time go?

- Technique using `xlc` compiler for an executable called `a.out`:
- Compile and link using `-pg` flag.
- Run `a.out`. The executable produces the file `gmon.out` in the same directory.
- Run several times and rename `gmon.out` to `gmon.1, gmon.2, etc...`
- Execute: `gprof a.out gmon.1 gmon.2 > profile.txt`

Profiling: gprof output

- Output may look like this:

% cumulative	self	self	self	total	
time	seconds	seconds	calls	ms/call	ms/call name
72.5	8.10	8.10	160	50.62	50.62 .snswp3d [3]
7.9	8.98	0.88			__vrec [9]
6.2	9.67	0.69	160	4.31	7.19 .snnext [8]
4.1	10.13	0.46	160	2.88	2.88 .snneed [10]
3.1	10.48	0.35	2	175.00	175.00 .initialize [11]
1.8	10.68	0.20	2	100.00	700.00 .rtmain [7]
1.5	10.85	0.17	8	21.25	1055.00 .snflwxyz@OL@1
0.7	10.93	0.08	320	0.25	0.25 .snxyzbc [12]

Profiling Techniques

- Look for the routine taking the largest percentage of the time. That is the routine, most possibly, to optimize first.
- Optimize the routine and re-profile to determine the success of the optimization.
- Tools on other machines: prof, gprof, apprentice, prism.

Assembly Code

- Being able to *read* assembly code is critical to understanding what a program is doing.
Writing assembly code is often unnecessary, however.
- To get useful assembly code on Blue Horizon, compile with the “-qsource” and -qlist” options.
- After being compiled, the output gets put in a “.lst” file.

Reading .lst Files

- At the top of the file, there is a list of line numbers. Find the line number(s) of the inner loop(s) of your program, then scroll down to where those lines appear (in the leftmost column).
- If you are using timers around your inner loop, it will usually be between the timing statements.

Don't Panic!

- There are a few commands that one wants to learn. They appear in the **third column** and they describe what the program is doing. If there are “unnecessary commands,” the program is wasting time.
- Additionally, there are “predicted” numbers of cycles in the fifth column. Determining how well these match up with the actual number of cycles per iteration is very useful.

Basic PowerPC Commands

- **fadd** = floating-point add
- **subf** = floating-point subtract
- **lfd** = load double word
- **lwz** = load integer word
- **stw** = store integer word
- **bc** = branch on count
- **addi** = add immediate
- **ori** = or immediate

More Information

PRISM Documentation:

<http://docs.sun.com:80/ab2/coll.514.2/PRISMUG/>

Parallel Communication Benchmarks:

<http://www.cse.ucsd.edu/users/baden/cse268a/PA/pa1.htm>

Timer Documentation:

man [gprof, prism, MPI_Wtime, etc...]