Performance Measuring on IBM SP-class Systems: Timing and Profiling

Performance Measuring with Timings: Wallclock

- Wallclock time (real time, elapsed time)
  - High resolution (unit is typically 1 μs)
  - 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.

Performance Measuring with Timings

- 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*, *not* the average times.

Wallclock Time

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

gettimeofday()

```
#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 (Tps, 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 = STARTTIME)

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. For instance, with the times in microseconds:
- (([program time]-[initialization time]) * [clock speed in Hz])/[number of cycles]

More About Timing

- Compute time of 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.

Profiling

- 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:

% time	cumulative seconds	self seconds	calls	self ms/call	total 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 routing 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, gvprof, apprentice, prism.