COMP3320/COMP6464: Performance Measurement, Benchmarking & Modelling

Alistair Rendell

See: *High Performance Computing*, Dowd and Severance
Chapter 6:Timing and Profiling
Chapter 15: Published Benchmarks
Chapter 16: Own Benchmarks

4.1 (Academic) Scientific Code Development

Scientific codes are invariably complex

- First: write the simplest possible code ensuring that it is functionally correct
  - explore the utility of what you have done
  - use the code to do science and publish papers!
- Second: rewrite the code attempting to minimize operation count and resource usage
  - do more calculations and publish more papers
- Finally: analyse performance and tune code
  - do grand challenge calculation and publish paper in Nature!

4.2 Performance Measurement

People begin to consider performance when:

- Their workload has increased and they need to squash 25 hours of computing into 24 hours
- The workload of others using the same computing resource has increased
- They are considering the purchase of a new computer and need to assess its performance

First we need to know the meaning of time and how to measure it!

4.3 Measuring Time

- Which time to use : wall time (elapsed time), or process time?
- Reliability issues (nb. typically time slice interval is $t_S \approx 0.01s$):

time:	wall	process
timer resolution t_R:	high √	low $(= t_S)$ ×
timer call overhead t_C:	low √	high ×
effect of time slicing / interrupts:	high ×	lower √
appropriate timing interval t_I:	$< 1t_S$	$> 100t_S$

- Error in $t_I \leq |\pm 2t_R + t_C|$ (may be variability in t_C; $t_I \leq 2t_R + t_C$ safer)
  - how to minimize these effects?
- Estimating t_R from (differences between) repeated calls to a timer function:
  - ☛ `1.5e-05 0 5.0e-6 0 0 0 0 5.0e-6 0 0 0 0 5.0e-6 0 ...` : $t_R \approx$ 5.0e-6, $t_C < t_R$
  - ☛ `2.0e-05 1.0e-6 7.0e-7 1.3e-6 1.0e-06 7.0e-07 ...` : $t_R \approx$ 0.3e-6, $t_C \approx$ 0.7e-6
  - ☛ `1.6e-05 1.1e-6 9.0e-7 1.0e-6 9.0e-7 9.1e-3 ...` : $t_R \approx$ 1.0e-7, $t_C \approx$ 1e-6
- nb. a low t_R means a 'high (degree of) resolution'

4.4 Scales of Timings

- Whole applications
- Critical 'inner loops'
  - how to identify these?
- Time for basic operations — eg. +, *
  - multiples of clock cycle
- Machine cycle time
  - 1GHz clock equivalent to 1nsec (10^{-9}s)

4.5 Total Program Timing

C, Korn and Bourne shell provide the time and timex utility

```
me@iwaki> time myprogram              ! This is under tcsh
  1.60u 0.19s 0:03.33 53.7%
me@iwaki> time myprogram              ! This is under csh
  2.0u 0.0s 0:03 65% 0+0k 0+0io 0pf+0w
me@iwaki> /usr/bin/timex myprogram
  real        2.98
  user        1.70
  sys         0.18
```

- For parallel programs on multi CPU machines user time can exceed elapse time
- High system time may indicate memory paging and/or I/O
- Ratio of user+system time to elapsed time can reflect other users on the machine
- Under csh we get information on average memory usage (program text+data structures), I/O operations (input+output), and page faults and swaps

4.6 Manual Timing: Routines

call date_and_time(date, time, zone, values)

```
#include <time.h>
#include <sys/times.h>
#include <unistd.h>
#include <sys/time.h>
int main(int argc, char **argv){
  struct tms cpu;
  struct timeval tp;
  struct timezone tzp;
  long tick;
  tick = sysconf(_SC_CLK_TCK);
  printf(" Ticks per second %ld \n",tick);
  times(&cpu);
  printf(" User   ticks %d \n",cpu.tms_utime);
  printf(" System ticks %d \n",cpu.tms_stime);
  gettimeofday(&tp, &tzp);
  printf(" Elapsed secs %d usec %d \n",tp.tv_sec,tp.tv_usec);
}
```


Manual Timing: Issues

Resolution (and overhead): You should have some idea of its value

- In some cases it may not be what is reported in a man page, eg it may say microseconds (1e-6) but are all the digits meaningful
- Often the resolution of the CPU timer is relatively low - one hundreth of a second is common

CPU Time: Take care with the meaning of CPU time. Some timing routines switch from CPU to elapsed time if the program is running in parallel

Baseline: Timing provides a baseline from which to judge performance tuning or comparative machine performance

Placement: How do we know where to place timing calls!

- Unix provides us with a number of tools to help with this

4.7 Subroutine Profiling: *prof*

- *prof* is the most common UNIX profiling tool. It is an extension of the compiler and linker environment. Provides a breakdown of how much time is spent in each routine of your code
- *prof* works by periodically sampling the program counter. In this sense it is statistical. The sampling rate (typically 0.01s) will determine the accuracy of the profile. Note greater sampling rates give higher overheads
- Must compile with "-p" profiling flag. Runtime generates a "mon.out" file containing profiling data that is post analysed.

```
me@iwaki> cc myprogram.c -p -o myprogram
me@iwaki> ./myprogram
me@iwaki> prof ./myprogram
 %Time Seconds Cumsecs  #Calls   msec/call  Name
  81.5    2.51    2.51       1   2510.      main_
  10.4    0.32    2.83       1    320.      sub2
   8.1    0.25    3.08       1    250.      sub1
                 ----- etc -----
```


Subroutine Profiling: *gprof*

- *gprof* is similar to *prof* (use -pg and gprof instead of -p and prof) but it also provides a call graph detailing who called who and how much time was spent in a given calling branch

```
                               called/total      parents
index  %time    self  descendents  called+self    name           index
                               called/total      children
                2.50        0.89      1/1          main  [2]
[1]    100.0    2.50        0.89      1          main   [1]
                0.27        0.31      1/1          sub1   [5]
                0.31        0.00      1/2          sub2   [4]
----------------------------------------------
                0.31        0.00      1/2          main   [1]
                0.31        0.00      1/2          sub1   [5]
[4]     18.3    0.62        0.00      2          sub2   [4]
----------------------------------------------
                0.27        0.31      1/1          main   [1]
[5]     17.1    0.27        0.31      1          sub1   [5]
                0.31        0.00      1/2          sub2   [4]
```


4.8 Basic Block Profiling

Often require finer detail than just subroutine based. Basic block profilers can provide information on time spent executing individual lines of code.

tcov: available on Sun and SPARC machines. Provides number of times each line was executed (not the time taken)

codecov: available with Intel compilers. How much of an application code is executed for a given workload.

collect/analyzer: is a recent addition to Sun workshop environment providing prog/gprof (without the need for a compilation flag) and line by line profiling (requires -g compiler flag). This product will be extended to provide data on cache misses, prefetch etc

performance counters: all of the main stream microprocessors are equipped with hardware registers that count operations. Libraries are provided to access this information.
PAPI (`http://icl.cs.utk.edu/projects/papi`)
PCL (`http://www.kfa-juelich.de/zam/PCL`)

4.9 Other Profiling Tools

- The profiling tools introduced above are not specific to C, they work with other languages such as Fortran
- We have concentrated on CPU performance, there are other useful tools
  - vmstat, vm_stat, memvis - virtual memory and CPU statics
  - mpstat, mpvis - parallel memory/CPU statics
  - netstat, nfsstat, nfsvis - network status and statistics
  - iostat, dkvis - I/O statics
  - sar - system activity report
  - top, prstat - list of most active processes
  - systat - system activity report
  - lockstat - kernel lock statics
  - dkstat - file status information

4.10 Benchmarking

Motivation

- Which machine is the fastest (for my applications)?
  - no simple answer: generally any single benchmark tests a single aspect of a machine
- To justify the purchase of a particular machine (eg APAC ran benchmarks)

What benchmarks

- Use some third party benchmark
- Create your own benchmark representing your own application mix
  - typically has larger program / data sizes
  - generally, will expose new (or a different mix of) features of the algorithm-architecture interaction
  - exposes the effects of compilers

4.11 Standard Benchmarks

- Simple performance measures:
  - MIPS: Millions of Instr'ns Per Second
    - Requires a notion of a 'standard' instr'n set, eg. VAX MIPS
  - MFLOPS: Millions of Floating Point Operations Per Second
    - Peak: the speed that you are guaranteed never to exceed!
    - Sustained: the rate achieved by a "standard" computation, eg. LINPACK (http://www.top500.org/lists/linpack.html)

  How meaningful are these? (the 'MachoFLOPS myth')
- Benchmark suites: a set of small benchmark programs
  - eg. SPECint/fp 2000 (http://www.spec.org), TPC (database; http://www.tpc.org/), STREAM (memory bandwidth; http://www.cs.virginia.edu/stream)

  √ Gives some measure of 'all-round' performance

  × Vendors play the benchmark game; users often disappointed

4.12 Top 500

Rank	Site	Computer	Processors	Year	R_{max}	R_{peak}
1	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution IBM	212992	2007	478200	596378
2	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution IBM	65536	2007	167300	222822
3	SGI/New Mexico Computing Applications Center (NMCAC) United States	SGI Altix ICE 8200, Xeon quad core 3.0 GHz SGI	14336	2007	126900	172032
4	Computational Research Laboratories, TATA SONS India	EKA - Cluster Platform 3000 BL460c, Xeon 53xx 3GHz, Infiniband Hewlett-Packard	14240	2007	117900	170880
5	Government Agency Sweden	Cluster Platform 3000 BL460c, Xeon 53xx 2.66GHz, Infiniband Hewlett-Packard	13728	2007	102800	146430

4.13 Standard Performance Evaluation Corporation: SPEC

spec **Standard Performance Evaluation Corporation**

home | benchmarks | results | contact | site map | site search | help

Benchmarks

- CPU
- Graphics/Workstations
- MPI/OMP
- Java Client/Server
- Mail Servers
- Network File System
- Power and Performance
- SIP
- Virtualization
- Web Servers

- Results Search

- Submitting Results
  CPU/Java/Mail/Power/SFS/Web
  MPI/OMP
  SPECapc/SPECviewperf

The Standard Performance Evaluation Corporation (SPEC) is a non-profit corporation formed to establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers. SPEC develops benchmark suites and also reviews and publishes submitted results from our member organizations and other benchmark licensees.

What's New:

01/27/2008 - SPEC has issued a call for papers for the SPEC International Performance Evaluation Workshop 2008, bringing together researchers and

See http://www.spec.org

4.14 Performance Modelling

- Accurate performance models are needed to understand / predict performance
- Given a problem size n, typically the execution time is $t(n) = O(n^2)$
  - challenge of HPC is in large n, not in complexity of $t(n)$
  - often (eg. vector operations) $t(n) = a_0 + a_1 n$; the values of a_0, a_1 are important!

    ie. $O(t(n))$ (tight upper bound), $\Omega(t(n))$ (lower), $\Theta(t(n))$ (upper+lower) concepts are inadequate
- A useful measure is the execution rate:

$$R(n) = \frac{g(n)}{t(n)}$$

where $g(n)$ is the algorithm's 'operation count', $g(n) = \Theta(t(n))$

  - e.g. graph of $R(n) = \frac{n}{10+n}$
  - note: if $g(n) = cn$, $a_0 =$ the startup cost, $c/a_1 = R(\infty) =$ the asymptotic rate
  - startup costs can be large, especially on vector computers
  - can use regression to determine a_0, a_1 by *measuring* $t(0), t(1000), \ldots$

4.15 Parallel Performance Modelling: Amdahl's Law#1

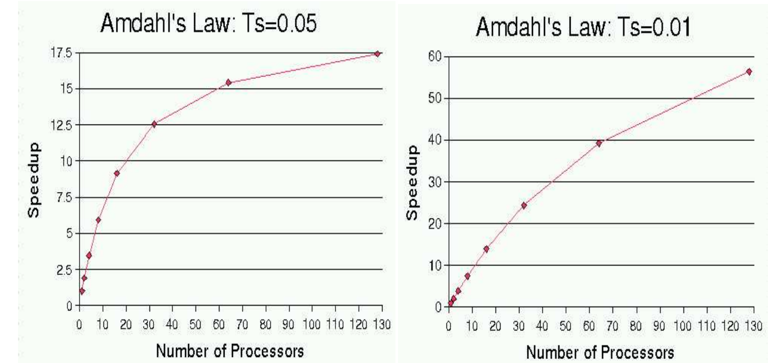
- The bane of parallel (||) HPC?
- Given a fraction f of 'slow' computation, at rate R_s, and R_f being the 'fast' computation rate:

$$R = (\frac{f}{R_s} + \frac{1-f}{R_f})^{-1}$$

- Interpreted for vector processing:
  - f is the fraction of unvectorizable computation, with R_f (R_s) being the vector unit (scalar unit) speed
- Interpreted for parallel execution with p processors:
  - f is the fraction of serial computation, with $R_f = pR_s$, ie.:

$$R_p = (f + \frac{1-f}{p})^{-1} R_s$$

4.16 Amdahl's Law#2: Speedup Curves



"Better to have two strong oxen pulling your plough across the country than a thousand chcickens. Chickens are OK, but we can't make them work together yet"

Amdahl's Law#3

- Other useful measures:
  - Speedup: $S_p = \frac{t_1}{t_p}$

    t_1 for the *fastest* serial algorithm, t_p is || execution time
  - Efficiency: $E_p = \frac{S_p}{p}$

    ideally $E_p = 1$; is $E_p > 1$ possible?
- Consequences:
  - for a given fixed f, there will be a limit to p that can be usefully applied, eg. $p \leq \frac{1}{f}$
  - this set back || computing 15 years!
- Counter notion: scalability
  - for a large p, only makes sense to use large n, ie. $n = n_1 p$
  - typically $f(n) = c'/n$, hence:

    $R(n) = R(n_1 p) = (\frac{c'}{n_1 p} + \frac{1-c'/(n_1 p)}{p})^{-1} R_s \approx \frac{p}{c'/n_1+1} R_s$
  - ie. $R(n)$ can increase linearly with p under these conditions

    $\Rightarrow$ || processing can be worthwhile!