Overview

- What to measure?
- Timers
- Benchmarking
- Profiling
- Finding hotspots
- Profile-guided compilation
- Messaging and network performance analysis
- Parallel performance analysis
- Further reading
What to Measure?

- *Scire est Mensurare* ("knowledge means measurement") - J. Keppler
- Types of measurements to help draw performance conclusions
  - Measuring elapsed execution time of (part of) a program
    - Which algorithm runs faster in practice?
  - Number of instructions executed, absolute total or rate per second
    - Compare to theoretical peak performance e.g. instructions per cycle (IPC)
  - Rate of floating point operations per second (Mflops)
    - Compare to theoretical or sustained Mflops peak performance
  - System events, e.g. cache hits, system calls
    - Determine how much and when/where overhead occurs
  - Communication latency
    - Reorganize communications and computation (improve latency hiding)
  - Parallel speedup
    - How effective is parallelism?
  - ...

9/22/10
Measuring Elapsed Time: Command-line Timers

- Measuring elapsed time with simple command-line timers
  - Unix/linux `time`
  - Sun Solaris `timex, ptime`

```
(time ./sqblas
...
32.54u 0.06s 0:32.63 99.9%
```

```
(ptime ./sqblas
... real 32.570
      user 32.498
      sys  0.064
```

Real time: the elapsed wall clock time

CPU user time: time spent in user-space

CPU system time: time spent in system space (OS kernel calls)
Measuring Elapsed Time: Real, User, and System Time

- **Real time** (or *wall clock time*) is the total elapsed time from start to end of a timed task

- **CPU user time** is the time spent executing in user space
  - Does not include time spent in system (OS calls) and time spent executing other processes

- **CPU system time** is the time spent executing system calls (kernel code)
  - System calls for I/O, devices, synchronization and locking, threading, memory allocation
  - Typically does not include process waiting time for non-ready devices such as disks

- **CPU user time + CPU system time ≤ real time**
  - CPU percentage spent on process = $100\% \times (\text{user} + \text{system}) / \text{real}$
Measuring Elapsed Time: System Timers

- **times()**
  - Gives user and system time in CPU’s CLK_TCK’s of a second

- **getrusage()**
  - Gives usage information of a system resource
  - Gives CPU usage in seconds and \( \mu s \ (\pm \Delta \mu s) \)

- **gettimeofday()**
  - Gives wall clock time in seconds and \( \mu s \ (\pm \Delta \mu s) \)

- **gethrtime()** (Solaris)
  - Gives wall clock time in nanoseconds

- **GetSystemTime()** (MS Windows)
  - Gives wall clock time in ms

- **MPI_Wtime()**
  - Wall clock time in double floating-point number of seconds
  - Part of MPI libraries
  - Resolution varies
Measuring Elapsed Time: Hardware Timers

- RDTSC x86 instruction
  - The 64bit “read time stamp counter” counts CPU cycles
  - Multicore systems have a RDTSC per core and these are not synched: must ensure thread affinity to use it reliably!

```c
__inline__ uint64_t rdtsc() {
  uint32_t lo, hi;
  __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
  return (uint64_t)hi << 32 | lo;
}
```

```c
uint64_t t0, tf;
t0 = rdtsc();
... code to time ...
tf = rdtsc();
printf("Time in CPU clocks = %lu\n", tf-t0);
```

Assumes that there is no rescheduling of this thread to another core
Measuring Elapsed Time: Benchmarking

- Measuring average elapsed time of (part of) a program using a spin loop
  - A spin loop is necessary when timer resolution is poor
- Can add another outer loop to compute standard deviation (variance) of the average elapsed time, which should hopefully be low
- Additional considerations
  - Cold start: include the first run in the timing result
  - Warm start: do not include the first run in timing result, caches are “warmed up”

```c
    t0 = start_time();
    for (run = 0; run < maxruns; run++)
        ... code to time ...
    tf = end_time();
    avg_exec_time = (tf - t0) / maxruns;
```

Example spin loop
Benchmarking: Measuring Mflops

- Estimate $Mflops$ for algorithm $A$ that requires $f(n)$ floating point operations for data size $n$:
  \[ Mflops = f(n) \times 10^{-6} / t_{sec} \]

- Inaccurate when the timer resolution is poor
  - Example: resolution of 10ms requires $t_{sec} \geq 1s$ to get double digit accuracy

- Run algorithm $A$ repeatedly until sufficient time $t_{sec}$ has elapsed, where $k$ is the number of runs, then calculate
  \[ Mflops = kf(n) \times 10^{-6} / t_{sec} \]

```c
maxruns = min_start_runs;
do
{ t0 = start_time();
   for (runs = 0; runs < maxruns; runs++)
      ... code to time ...
   tf = end_time();
   maxruns = 2*maxruns;
} while (tf - t0 < min_time);
avg_exec_time = (tf - t0) / runs;
```
Profiling

- **Instrumentation-based (event-based) profiling**
  - Instruments code to record events, usually with counters
  - Disadvantage: high overhead that distorts elapsed time
    - Intel *codecov*

- **Sampling-based profiling**
  - Uses system interrupt to take snapshots of process’ point in code
  - Advantage: very low profiling overhead
  - Disadvantage: low statistical accuracy and possible timing anomalies
    - GNU *gprof*
    - Microsoft *perfmon.exe*
    - Intel VTune
    - Sun Studio profiler *collect -p*

- **Hardware counter (event-based) profiling**
  - Uses special CPU registers to count CPU events
  - Advantage: more detailed information with no profiling overhead
    - Intel VTune
    - Sun Studio profiler *collect -h*
Instrumentation-Based Profiling

- Adds instrumentation to code to record events with counters
  - During execution of the program the counters are updated
  - Counter information is saved to a profile (some data file(s))
  - The profile is displayed for inspection
- “Heisenbugs” phenomenon
  - Occurs when profiling an application using a profiling technique that changes the running time
  - The internal timing of events in the applications changes
  - When trying to find timing-related bugs, they may no longer occur
Intel Codecov

- Compile: `icc -prof-genx -o prog ...
- Run: `prog`
- Combine generated .dyn file into profile: `profmerge`
- Convert to readable format: `codecov -counts`

The ^ denote execution counts
Sampling-Based Profilers

- Sampling-based profilers are statistical profilers
  - Takes periodic sample of position in code (program counter)
  - Some profilers use instrumentation to track number of calls and record information on the calling targets to build a call graph
  - Execution time of function \( f \) = total running time / \# calls to \( f \)

- Flat profilers produce run time statistics for each function and subroutine
  - Sampling time adds up to 100%

- Call-graph profilers produce hierarchical run time statistics
  - Flat profile information is propagated along the edges of the call graph to the parents (the callers)
  - Time spent is then added to time of the caller
  - Caller-callee statistics are also produced
Flat Versus Call-Graph Profiles

main calls A, A calls B, B returns, A calls C, C returns, A calls B, etc…

- Flat profile
  - Time(A) = 2/7 = 29%
  - Time(B) = 5/7 = 71%
  - Time(C) = 0
  - Time(D) = 0

- Call graph profile
  - # calls to A = 2
  - # calls to B = 5
    - # calls from A to B = 3/5
    - # calls from D to B = 2/5
  - # calls to C = 5
    - # calls from A to C = 3/5
    - # calls from D to C = 2/5
  - # calls to D = 1
Profiling with gprof

- The gprof Unix/Linux profiler is a sampling-based profiler
- Typical sample rate is 10ms
- Uses instrumentation for call graph profiling
- Steps:
  - Compile with `gcc -pg -o prog`
  - Run the program `prog` to generate `gmon.out`
  - Run `gprof prog gmon.out`

### Example flat profile

<table>
<thead>
<tr>
<th>time</th>
<th>seconds</th>
<th>cumulative</th>
<th>seconds</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>76.52</td>
<td>45.35</td>
<td>45.35</td>
<td>205340</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>block_mult</td>
<td></td>
</tr>
<tr>
<td>23.24</td>
<td>59.13</td>
<td>13.78</td>
<td>37</td>
<td>0.37</td>
<td>0.37</td>
<td>0.37</td>
<td>sqmat_check</td>
<td></td>
</tr>
<tr>
<td>0.32</td>
<td>59.32</td>
<td>0.19</td>
<td>136444</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>sqmat_mult</td>
<td></td>
</tr>
<tr>
<td>0.10</td>
<td>59.38</td>
<td>0.06</td>
<td>2</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>sqmat_rand</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>59.38</td>
<td>0.00</td>
<td>1709</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>cputime</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>59.38</td>
<td>0.00</td>
<td>37</td>
<td>0.00</td>
<td>1.23</td>
<td>1.23</td>
<td>sqmat_mflops</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>59.38</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>timeres</td>
<td></td>
</tr>
</tbody>
</table>
## Profiling with gprof

**Example call graph profile**

<table>
<thead>
<tr>
<th>index</th>
<th>called</th>
<th>name</th>
<th>index</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>72384</td>
<td>72384</td>
<td>sym_id_parse [54]</td>
<td>1508</td>
<td>1508</td>
<td>cg_dfn [15]</td>
</tr>
<tr>
<td>4/9052</td>
<td></td>
<td>cg_tally [32]</td>
<td>1508</td>
<td>1507/1507</td>
<td>cg_assemble [38]</td>
</tr>
<tr>
<td>6032/9052</td>
<td></td>
<td>propagate_flags [52]</td>
<td>1509</td>
<td>1509/1509</td>
<td>is_numbered [9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1508/1508</td>
<td>propagate_time [14]</td>
</tr>
<tr>
<td>5766/5766</td>
<td></td>
<td>core_create_function_syms [41]</td>
<td></td>
<td>1508/1508</td>
<td>post_visit [12]</td>
</tr>
<tr>
<td>1513/1537</td>
<td></td>
<td>core_create_function_syms [41]</td>
<td></td>
<td>1508/1508</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1505</td>
<td>print_line [16]</td>
</tr>
<tr>
<td>1511/1511</td>
<td></td>
<td>core_create_function_syms [41]</td>
<td></td>
<td>2/9</td>
<td>print_name_only [25]</td>
</tr>
<tr>
<td>2/1510</td>
<td></td>
<td>arc_add [31]</td>
<td></td>
<td>1430/1430</td>
<td>core_create_function_syms [41]</td>
</tr>
<tr>
<td>1508/1510</td>
<td></td>
<td>cg_assemble [38]</td>
<td></td>
<td></td>
<td>source_file_lookup_path [17]</td>
</tr>
<tr>
<td>[8]</td>
<td>1510</td>
<td>arc_lookup [8]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1509/1509</td>
<td></td>
<td>cg_dfn [15]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1508/1508</td>
<td></td>
<td>propagate_flags [52]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[10]</td>
<td>1508</td>
<td>inherit_flags [10]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1508/1508</td>
<td></td>
<td>cg_dfn [15]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1508/1508</td>
<td></td>
<td>cg_dfn [15]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9/22/10
Statistical Inaccuracy of Sampling-Based Profiling

- Statistical inaccuracy of sampling occurs when
  - A function executes for a very small amount of time
  - Is called infrequently
  - Is called periodically, but the period has high variability

- Can improve statistics by combining sampling outputs:
  1. `mv gmon.out gmon.sum`
  2. Run prog as many times as needed and each time merge stats: `gprof -s prog gmon.out gmon.sum`

Example showing the statistical inaccuracy of measuring the calling time of C with samples
Sampling-Based Profiling with Sun Studio Profiler

- Takes sample of the subroutine call stack at predetermined sample rate
  - 1ms (collect -p hi), 10ms (collect -p on), 100ms (collect -p lo)
- Collects calling time of each function via event mechanism (see next slide)
  - Sun Studio: collect -p on -o sample.er ./sqmatb

At this point in time: main called sqmat_flops called sqmat_mult called block_mult
Sampling-Based Profiling with Sun Studio Profiler

- Flat profile with user, system, and wall clock time
- *Inclusive time* shows the time spent in function and the functions called by it
- *Exclusive time* shows the time spent in the function by excluding the callers
Hardware Counter Profiling

- CPU hardware counters record CPU events
  - Number of instructions executed
  - Number of memory loads/stores
  - Cache misses, TLB misses, branch mispredicts
  - Etc …

- Limited number of such counters are available
  - Counters can be configured to track a particular class of events
  - UltraSparc IIIi has two counters
  - For example: one counter can be set to count # instructions while the other counts # D-cache misses

- Hardware counter overflow causes interrupt
  - Interrupt saves the counter value in profile
  - Overflow value can be set
  - Should be low to minimize interrupt overhead
HW Counter Profiling with Sun Studio Profiler

- Works on Solaris and selected Linux platforms
- To obtain a list of available HW counters: `collect`
- Steps:
  - Compile: `cc -g -o prog ...`
  - Compile with Fortran: `f95 -g -o prog ... -lfcollector`
  - Profile: `collect -o prog1.er -p on -h ctrl1,on,ctr2,on prog`
  - Multiple runs (different HW counters): `collect ... -g profile.erg`
  - Analyze: `analyzer prog1.er`
- Examples
  - Count instructions: `collect -h cycles,on,insts,on ...`
  - FP operations: `collect -h fpadd,on,fpmul,on ...`
  - D and E cache misses: `collect -h dcstall,on,ecstall,on ...`
- Can also be run from the Sun Studio Express IDE
Example

```
collect -o test.memory.1.er -S off -p on -h ecstall,on,cycles,on ./cachetest -g memory.erg
collect -o test.memory.2.er -S off -h icstall,on,dcstall,on ./cachetest -g memory.erg
```
Iterative Optimization: the Profile + Improve Cycle

- Iterative optimization
  1. Check for correct answers (program must be correct!)
  2. Profile to find the hotspots, e.g. most time-consuming routines
  3. Optimize these routines using compiler options, compiler directives (pragmas), and source code modifications
  4. Repeat 1-3

- Optimizing the hotspots of a program improves overall performance

- Programs with “flat profiles” (flat timing histogram)
  - Programs with lots of routines that each take a small amount of time are difficult to optimize
Finding Hotspots

- Profilers help find “hotspots”
  - To find hotspots, run program and profiler for small, medium, and large data sets
  - Hotspots show a large percentage of a particular profiling statistic such as time or cache misses

- Hotspots for time
  - Parts of the program that are frequently executed
  - Parts of the program that are slow

- Hotspots for data cache misses
  - Parts of the program that are data intensive
Profile-Guided Optimization

Profile-guided optimization (PGO) collects execution profile statistics to enhance effectiveness of optimizations.
Profile-Guided Optimization

- PGO helps improve I-cache, paging, branch prediction
- Enhances optimizations:
  - Basic block ordering
  - Better register allocation
  - Better decision of functions to inline
  - Function ordering
  - Switch-statement optimization
  - Loop trip counts can enable more aggressive loop optimization
- What type of applications will benefit?
  - Applications with lots of functions, calls, or branches
  - Applications with computations spread throughout
  - Applications with consistent hot execution paths

Significant benefit  No benefit
Profile-Guided Optimization

- PGO is a three-step process:
  1. Compile code with instrumentation
  2. Run
  3. Recompile using performance statistics
- Example Sun compiler:
  - `suncc -xprofile=collect:data -o prog ...
  - `prog`
  - `suncc -xprofile=use:data -xlinkopt -o prog ...
- Example Intel compiler:
  - `icc -prof-gen -prof-dirc:data -o prog ...
  - `prog`
  - `icc -prof-use -prof-dirc:data -ipo -o prog ...`
CPU Bound versus I/O Bound Processes

- A *CPU bound* process is compute intensive
  - Very few I/O operations
  - Execution speed of algorithm is determined by CPU
  - Use performance profiling

- An *I/O bound* process is I/O intensive
  - Process includes I/O operations such as file access, message passing over network
  - Execution speed is limited by system’s I/O latencies
  - Performance analysis method depends on I/O operations: operations on file on disk, messages over pipes and sockets, etc.
Measuring Message Passing Performance

- Measuring message latency and bandwidth $t_{comm} = t_{startup} + w t_{data}$ using ping-pong
  1. First node sends message to second node
  2. Second node sends message back
  3. Repeat 1-2 and take average time
- Message sizes are chosen at regular intervals to cover a range of message sizes
- Message startup time can be inferred from the above or measured with empty messages

- Network topology considerations
  - Message latency depends on destination, e.g. number of node hops

- Performance measurements of global communications
  - Either implemented with point-to-point messaging
  - Or uses message broadcast: should measure the elapsed time until all receivers received the message
Time Complexity of a Parallel Program

- Parallel computation time: \( t_{\text{comp}} \)
- Parallel communication time for sequence of \( N \) messages:
  \[ t_{\text{comm}} = N(t_{\text{startup}} + w \ t_{\text{data}}) \]
- Example with three different values of \( t_{\text{comm}} \)
  - Sequential time is fixed = 1000
  - Assume \( P \) processors
  - Parallel speedup:
    \[ S_P = 1000/(t_{\text{comp}} + t_{\text{comm}}) \]
  - Parallel computation time:
    \[ t_{\text{comp}} = 1000/P \]
  - Constant communication time:
    \[ t_{\text{comm}} = 1 \]
  - Constant communication time:
    \[ t_{\text{comm}} = 10 \]
  - Communication time has \( P \) steps:
    \[ t_{\text{comm}} = P+1 \]
Performance Prediction Graph

- Performance prediction graph is a log-log graph of $t_{comp}$ and $t_{comp}$
- Below a critical number of processors $t_{comm}$ is minimal
- Example with two $t_{comm}$
  - Suppose parallel computation time: $t_{comp} = 1000/P$
  - Case 1: constant communication time: $t_{comm} = 10$
  - Case 2: communication time has $P$ steps: $t_{comm} = P+1$
  - Communication is reasonably low (in both cases) only when $P \leq 10$ such that $t_{comp}/t_{comm} > 10$ gives almost linear speedup: $S_P = 1000/(t_{comp} + t_{comm}) \approx 1000/t_{comp} = P$
Overlapping Computation with Communication

- Performance prediction graph is a log-log graph of $t_{comp}$ and $t_{comp}$
- Assume we can perfectly overlap computation with communication: $t_P = \max(t_{comp}, t_{comm})$
- Although the timings are better, the speedup limits persists when $t_{comp} < t_{comm}$
Example: \[ A = \sum_{i=1}^{N} a_i \]

- A simple summation example

**Algorithm:**
- Each processor \( j \) performs a local summation \( a[(j-1)k+1..jk] \) where \( k \) is such that \( N = kP \)
- Each processor \( j \) then sends its sum \( A_j \) to P0
- P0 sums the \( A_j \)

**Time estimate (relative):**
\[ t_P = t_{\text{comp}} + t_{\text{comm}} \]
\[ = (N/P + P-1) + \gamma(P-1) \]
\[ = N/P + (\gamma+1)(P-1) \]
where \( \gamma \) is the number of additions that can be performed per numeric value transmitted over the network
Example: \[ A = \sum_{i=1}^{N} a_i \]

- Now using a \textit{reduction tree}

- Algorithm:
  - Each processor $j$ performs a local summation $a[(j-1)k+1..jk]$ where $k$ is such that $N = kP$
  - Repeat until root of tree has the sum:
    - Each processor sends sum to parent in tree (could be self)
    - Parent sums the two values

- Time estimate (relative):
  \[ t_p = \frac{N}{P} + (\gamma+1)\log_2 P \]
  where $\gamma$ is the number of additions that can be performed per numeric value transmitted over the network
Example: \[ A = \sum_{i=1}^{N} a_i \]

Simple sum

\[ t_{\text{comp}} = \frac{10000}{P} + P - 1 \]

\[ t_{\text{comm}} = 9(P-1) \]

Reduction tree

\[ t_{\text{comp}} = \frac{10000}{P} + \log_2 P \]

\[ t_{\text{comm}} = 9 \log_2 P \]

Performance prediction graphs
Example: \[ A = \sum_{i=1}^{N} a_i \]

Simple sum

Reduction tree

\[ \text{Speedups } S_P = \frac{t_s}{t_P} = \frac{N}{t_P} \text{ with } N=10000 \text{ and } \gamma=9 \]
Cost-Optimal Algorithms

- The cost (or work) of a computation is
  \[ \text{cost} = (\text{execution time}) \times P \]
- Serial computation: \( \text{cost} = t_s \)
- Parallel computation: \( \text{cost} = t_P P \)
- Parallel algorithm is cost optimal when there is some constant \( k \) such that \( \text{cost} = t_P P = k t_s \)
- In other terms, a parallel algorithm is cost-optimal when \((\text{parallel time complexity}) \times P = (\text{sequential time complexity})\)
- Examples
  - Cost-optimal: sequential complexity \( O(n \log n) \), parallel complexity \( O(n/P \log n) \)
  - Not cost-optimal: sequential complexity \( O(n^2) \), parallel complexity \( O(n^2/P \log n) \)
Visualization Tools

- **Space-time diagram** of a parallel program
  - Process (vertical axis) and time (horizontal axis)
  - Duration of events, such as computing, waiting, message passing
  - Massage sends and receives (arrows)
- May not show overlapped computation/communication (latency hiding)
  - Example: jumpshot tool for MPI (discussed later)

*Example space-time diagram of barrier synchronization (with jumpshot)*
Further Reading

- [HPC] pages 125-142
- [PP2] pages 62-75
- [SPC] pages 56-65
- Optional:
  - [OPT] pages 19-50
  - [SRC] pages 443-468