Lecture 8

Scalability and
Performance Measurement

Scalability

• Recall the isoefficiency function…
• … this function tells us how quickly serial work $W$ must grow as we increase $P$….
  so that the efficiency will remain constant
• We now consider scalability in greater detail
Overhead

- Let $W$ be the work required to carry out a computation, i.e. $W = T_1$
- In general $T_p \geq W/P$, or $W \leq P T_p$
- Define $T_o = PT_p - W$ as the total overhead or the overhead function
- We call $PT_p$ the cost, or the processor-time product
- Thus, $E_p = W / (P T_p) = W / \text{cost}$

Cost optimality

- A system is cost-optimal if cost of the parallel computation has the same asymptotic growth as fastest serial algorithm
- The cost should grow at the same rate as $W$, i.e. $PT_p = \Theta(W)$
- Thus, efficiency $\approx$?
An example

- Consider a serial algorithm running in time
  \[ W = n \log n \]
- Let \( T_p = (\log n)^2 \) on \( P=n \) processors:
  cost = \( PT_p = n(\log n)^2 > W \)
- The system solving this problem is not cost optimal

Why does efficiency decrease with \( P \)?

- Recall that efficiency
  \[ E_P \equiv \frac{T_1}{(PT_p)} = \frac{W}{PT_p} \]
- Plugging this into the overhead equation
  \[ T_o \equiv PT_p - W \text{ we have} \]
  \[ E_P \equiv \frac{1}{1 + T_o/W} \]
- If \( W \) remains fixed
  - Overhead \( (PT_p) \) often increases with \( P \)
  - Efficiency must therefore decrease
A non-cost optimal example

- Summing N numbers on N processors
  - $W = N - 1$
  - $T_N = \lg(N)$
- Cost = $N \lg N$
- Efficiency $= E_P \equiv 1/(1 + T_d/W)$

\[
= 1/(1 + (\text{Cost} - W)/W)
= 1/(1 + (N \lg N - N)/N)
= 1/(1 + \lg N)
= 1/ (\lg N) << 1
\]

A cost optimal variant

- Summing N numbers on $P < N$ processors
  - $W = N - 1$
  - $T_P = (N/P) + \lg(P)$
- Cost = $N + P \lg(P)$
- If we maintain $N = \Omega(P \log P)$, then system is cost optimal
Scalability

- We say that a system is **scalable** if we can maintain a (nearly) constant running time as we increase the work with the number of processors.
- Equivalently, a system if scalable if we can maintain a constant level of parallel efficiency.
- When we think about scalability we ask: “how quickly must the computational work grow with P?”

Relationship of scalability and cost-optimality

- Recall that a cost optimal system has an efficiency of $\Theta(1)$.
- A scalable system can be made cost-optimal if we grow the problem size with $N$.
- For example, with summation, we can maintain cost-optimality if we grow $N$ as $\Theta(P \lg P)$.
- The *isoefficiency function* tells us the required growth rate, and how scalable the system is.
- It vary among different systems.
Isoefficiency function

- How quickly must the workload grow, as a function of \( P \), in order to maintain a constant level of efficiency
- Consider the ODE solver
  - \( N \) = Problem size
  - \( P \) = Number of processors
  - Computational work = \( W = 3N (= T_1) \)

Isoefficiency function for the ODE solver

- Let a floating point operation take unit time
- Normalized message start time = \( \alpha \)
- Parallel running time \( T_p \)
  - Perfect parallelization of \( W \) + communication overhead \( W/P + 2\alpha \)
- Parallel efficiency \( E_p = T_1 / (PT_p) = 3N/(2\alpha P + 3N) \)
- Rewriting to obtain expression for \( N \) in terms of \( P \)
  - \( N = (2/3) \alpha P (E_p/(1 - E_p)) = O(P) \)
- So long as we can grow \( N \) with \( P \), then the system is scalable
A different problem

- A linear isoefficiency function a “nice” function in the sense that growth is reasonable
- But not all isoefficiency functions are nice
- The isoefficiency function for summation
  \[ N = \frac{2E_P}{1 - E_P} \alpha P \log(P) \]
- Summation isn’t as scalable as the ODE solver
  - If we increase the number of processors from 32 to 1024 (x32), we must increase the work by a factor 160
  - We may run out of memory
- Similarly, the ODE solver is not scalable if we check convergence after every time step

How did we come up with the isoefficiency function?

- \( E_P = W / P T_P = W / (W + T_o(W,P)) \)
  \[ = \frac{1}{(1 + T_o(W,P)/W)} \]
- Solving for \( W \)
  \[ W = \frac{E_P}{1 - E_p} T_o(W,P) = K T_o(W,P) \]
- This is the isoefficiency function, the required growth of \( W \) as a function of \( P \)
- For summation:
  \[ N = \frac{2E_p}{1 - E_p} \alpha P \log(P) \]
- In order for a system to remain cost-optimal as it is scaled up, we require that \( W = \Omega(f(P)) \), where \( f(P) \) is the isoefficiency function
Re-examining scaled speedup

- We define the speedup as
  \[ \frac{W}{T_p(W,P)} \]
- The scaled speedup (linear scaling) is
  \[ \frac{PW}{T_p(PW,P)} \]
- If we are scaling a problem according to the isoefficiency function \( \Theta(P \lg P) \), what is the scaled speedup?
  \[ \frac{PW \lg P}{T_p(PW \lg P, P)} \]

Scaling

- Plot the efficiency for the problem of adding n numbers on p processors
  - \( t_{add} = 10 \), time to communicate = 1
  - \( p = 1, 4, 16, 64, 256 \)
- Fixed workload
  - Let \( n=256, W = 255 \)
  - Speedup = \( W / T_p(W,P) \)
- Scaled workload, base case \( n=256, p=1 \)
  - \( W = \Theta(P) \)
  - Scaled speedup = \( PW / T_p(PW,P) \)
- “Isoefficient scaled workload”
  - \( W = \Theta(P \log P) \)
  - Isoefficient scaled speedup = \( PW \lg P / T_p(PW \lg P,P) \)
Results

- Running time for adding up n numbers of p processors:
  \[ \frac{n}{p} - 1 + 11 \log p \]

- Fixed workload
  - Let \( n=256 \)
  - \( W = 255 \)
  - Speedup = \( W / T_p(W,P) \)

Results

- Linear scaling
  - base case: \( n=256, p=1 \)
  - \( W = \Theta (P) \)
  - Scaled speedup \( PW / T_p(PW,P) \)

- Isoefficient scaling
  - \( W = \Theta (P \log P) \)
  - Isoefficient scaled speedup \( PW \log P / T_p(PW \log P,P) \)
Reporting and Displaying Performance

• Give the viewer sufficient information to…
  – Draw their own conclusions
  – Reproduce your results

• Tabulate and display the results fairly
  – Avoid misleading techniques
  – See the Bailey paper for examples of how not to display and report performance data

Measures of Performance

• Completion time for a given workload
• Throughput: amount of work that can be accomplished in a given
• Relative performance: given a reference architecture or implementation
Relative performance

- If $T_B = 1.5$, $T_A = 1.0$
  - “A is 1.5 times faster than B”
- **Execution time on machine B** = $T_B$
  Execution time on machine $A = T_A$
- But what about…
  - $T_A / T_B = 2/3$; A is 33% faster than B
- Adopt the convention of reporting a “speedup”

Challenges to measuring performance

- **Reproducibility**
  - Transient system operating conditions
  - Document systems or program configuration, as well as inputs
  - Dedicated access is often preferred
- **Measurements are imprecise**
  - “Heisenberg uncertainty principle;” measurement technique may affect performance
  - Variations in performance are inevitable; OK if we can explain and tolerate them
- **Explain anomalous behavior, but ignore anomalies that are insignificant**
Document the operating conditions

`uname -a`

```
Linux valkyrie.ucsd.edu 2.4.21-4.0.1.ELsmp #1
SMP Sat Nov 29 04:15:49 GMT 2003 i686 i686
i386 GNU/Linux
```

g++ -v

```
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/3.2.3/specs
Configured with: ../configure --prefix=/usr
--mandir=/usr/share/man
--infodir=/usr/share/info --enable-shared
--enable-threads=posix --disable-checking
--with-system-zlib --enable-__cxa_atexit
--host=i386-redhat-linux
```

Repeatability

- “I get different timings. What should I report?”
- Repeat results 3 to 5 times until at least 2 measures agree to within the desired tolerance (5%, 10%…)
- Report the best timings or the mean
- Also report extreme values
- A scatter plot or error bar can be useful
Measuring performance

- Measurement tools
  - System clocks
    - Often platform dependent, especially library routines, e.g. MPI_Wtime()
    - Unix time command does a reasonable job for long-running programs
  - Hardware performance monitors

- Measures of time
  - Elapsed time (wallclock time)
  - CPU time = system + user time
Complications

- Timer quantization
- Overheads in the measurement technique
- Cost of measuring a full run is prohibitive
  - Ignore startup code if you plan to run for a much longer time in production
- Transient behavior
  - Repeat your measurements
  - “Warm up”
  - Ignore outliers unless their behavior is important to you
  - Average time, maximum time, minimum time?

What’s wrong with MFLOP rates?

- Different algorithms employ different numbers of floating point operations, e.g. Strassen’s matrix multiply algorithm
- Different library implementations can execute different numbers of FLOPs, e.g. log()
- Precision affects timing
- Floating point operations take different times
  - Divide is much slower than multiply or add
  - Some machines have a fused multiply-add
- MFLOP rates ignore memory access time