Wall Clock Time

- The simplest performance measurement: "wall clock time"
- Elapsed time from beginning to end of program execution
- Not the best metric for performance analysis: includes several components beyond the control of the application developer
  - O/S overheads (starting job, reading files, ...)
  - Multi-process environment
    - daemons (web server NFS server, ...)
    - other users' jobs
- If you use this metric:
  - gather data on a "lightly loaded" system
  - take mean of several measurements

CPU Time

- A common Unix utility: `time`
- Usage:
  ```
  % time x
  ```
- Runs `x`, then prints elapsed time, CPU time in user process, CPU time in O/S routines on behalf of user process
  ```
  % time ls -l
  ...output from "ls"
  ```
- Same command, from tcsh (where time is a shell command):
  ```
  0.000u 0.020s 0:00.11 18.1%
  ```
Bandwidth
- A different way to describe performance is bandwidth
  - also known as “throughput”
- Useful for describing parallel systems or other situations where several tasks are executed
- Elapsed time (aka “latency”) describes the amount of time required for a single job
  - unit: $T$
- Bandwidth describes the number of jobs that can be completed per unit of time
  - unit: $1 / T$

Example
- Suppose we have an N-Body application, and we need to simulate a system with 10,000 bodies
- We know it runs fine for 10 bodies, but are concerned it won’t scale to 10K bodies
- Methods to improve execution time (latency):
  - better algorithm
    - PP: 50,000,000 force calculations per time step
    - B-H: $130,000 \times 3 = 390,000$ forces per time step
  - more efficient data structures
    - reduce constant, but not asymptotic efficiency
    - vectors provide better instruction level parallelism
    - also better locality (caching)

Example (cont’d)
- Parallelism allows another possibility: improved bandwidth
- A parallel algorithm breaks job into smaller pieces, works on the pieces in parallel
- Our overall goal is still to have our program finish sooner, but now we have another way to approach the problem:
  optimize number of pieces completed per unit of time
- Performance can be improved through better throughput
  - more processors
  - smaller messages
  - faster delivery of messages
  - fewer messages

Example (cont’d)
- Suppose our machine is a Linux cluster with 10 nodes
  - parallel algorithm might have 1,000 bodies/node
  - chordal ring: each process computes $1,000,000$ forces/msg*, or $5,000,000$ forces per time step
  - parallel B-H: depends on how tree partitioned (see Fox, et al)

* all $n \times n$ interactions with bodies arriving on a token
Parallel Performance

Wall clock time for the parallel program is a function of:
- partitioning (can program be split into $N$ even parts?)
- parallel programming overhead (making, sending messages)
- system overhead (time waiting for work)

\[
T_p = T_i + \text{overhead}
\]

Speedup

Speedup is a measure of how much faster the parallel version runs:

\[
S = \frac{T_i}{T_p}
\]

Example:

\[
T_i = 9.5 \text{ sec} \\
T_p = 3.6 \\
S = \frac{9.5}{3.6} = 2.63
\]

Amdahl’s Law

Speedup is limited by the portion of the application that is inherently sequential.

Let $p$ be the portion that is not parallelizable.

\[
S = \frac{T_s}{T_p} = \frac{T_s}{pT_s + (1-p)T_s/N}
\]

Amdahl’s Law (cont’d)

The limit to $S$, as the number of processors $N$ increases to infinity:

\[
S = \frac{T_s}{pT_s + (1-p)T_s/N} = \frac{T_s}{pT_s} = \frac{1}{p}
\]
Amdahl’s Law (cont’d)
- Example:
  - \[ p = 0.1 \]
  - \[ S = \frac{1}{p} = 10 \]

Massive Parallelism
- The term “massive parallelism” means the use of a very large number of processors
  - coined by groups working on SIMD machines
  - \( N > 1000 \)
  - origin of the name “MasPar”
- Is “massive parallelism” feasible, or cost-effective, given Amdahl’s law?
  - are there problems where \( p < 0.001 \)?
  - does overhead mean speedup of 1000 is impossible?

Massive Parallelism (cont’d)
- Major speedups are possible if one increases the problem size along with the number of processors
  - Gustafson: “you wouldn’t hire 1000 painters to paint a kitchen”
  - contains parts of Gustafson’s CACM paper “Reevaluating Amdahl’s Law”
- Describes simulation of 2D acoustic wave
- Speedups over 1000x on 1024-node NCUBE system
  - NCUBE: competitor to Intel’s iPSC hypercube

Measuring Performance
- \texttt{time} shell command
  - returns user time and system time for the process
- Unix library calls
  - O/S specific
  - call once to reset timer
  - call again to measure CPU time since previous call
- Example: \texttt{clock()}
  (code on next slide)
Measuring Performance (cont’d)

#include <time.h>
clock_t start = clock();
// ... code to measure ...
clock_t end = clock();
float elapsed =
   (float(end) - float(start))/CLOCKS_PER_SEC;
cout << "time: " << elapsed << endl;

Fine-Grain Measurements

- To improve single-processor performance, it’s necessary to know where the application is spending most of its time
- An execution profile is a table showing the amount of CPU time in each function
  - reported as time and percent of total
  - table shows time in the function plus cumulative time in called functions
- Many scientific applications follow the 90/10 rule:
  - 90% of time spent in 10% of code
    - as measured by locality of reference for i-cache
    - especially true of applications that are candidates for SPMD

Profiles

- Software that generates an execution profile typically uses PC sampling
  - interrupt the program periodically
  - every 100 instructions? 1000?
  - record location of program counter (CPU’s PC register)
  - write locations to a file
- Table produced by application that reads trace file
  - maps PC values back to source code function
- Slight overhead for sampling may skew results
- New software has lower overhead, doesn’t require linking with sampling library

clock() Measurements

- clock(), getrusage(), and similar functions measure application time plus time spent in parallel libraries (e.g. MPI)
- Does not measure time a process is blocked, e.g. while it is waiting for an MPI message
Performance Profiles for Parallel Programs

- For parallel programs, we want to know not only how time is spent within a process (one node of an SPMD program).
- We also need to know about interprocess communication:
  - Time spent composing messages
  - Waiting for messages
  - Synchronization (e.g., at barriers, or in collective operations)
- One method: software library records events, as in execution profile, and graphic utility displays time line of processes and their interaction.

Viewing Message Patterns

- A very early MPI-based tool: upshot
- Newer version, called jumpshot, uses Java

TAU

- TAU is a UO research project headed by Allen Malony
- TAU = Tuning and Analysis Utilities
  - aka "tools are us"
- Platform-independent software to measure, analyze performance
- Supports sequential, MPI, and OpenMP applications
- Tools are installed on p690
- Read on-line documentation to learn more