Embedded computing for scientific and industrial imaging applications

Lecture 13 - Parallel computing, Amdahl’s law

Outline

- Basic concepts
- Shared vs. distributed memory
- OpenMP (shared)
- MPI (shared or distributed)

Some references

- P. S. Pacheco, An Introduction to Parallel Programming, Elsevier, 2011.
- T. Rauber and G. Ruenger, Parallel Programming For Multicore and Cluster Systems, Springer, 2010.
- C. Lin and L. Snyder, Principles of Parallel Programming, 2008.
- L. R. Scott, T. Clark, B. Bagheri, Scientific Parallel Computing, Princeton University Press, 2005.

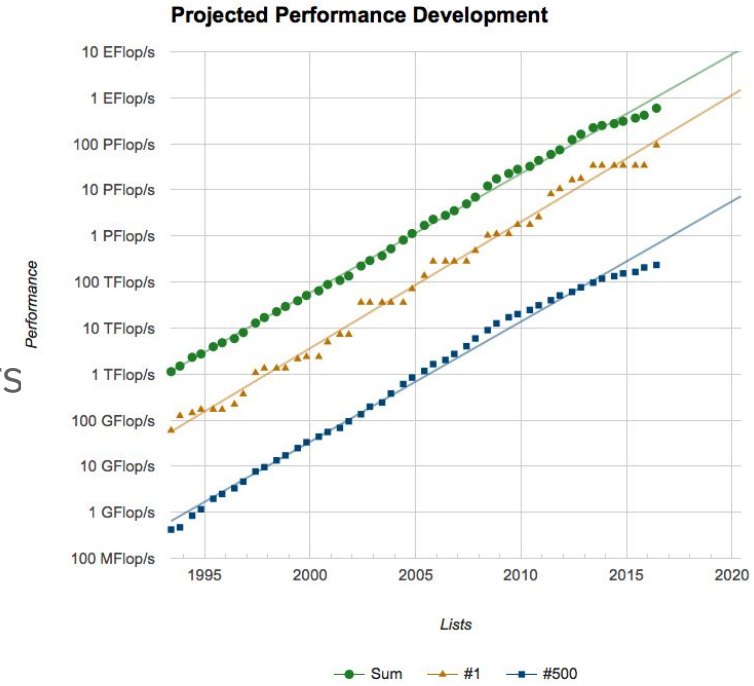
And more...

Increasing speed

Moore's Law: Processor speed doubles every 18 months.
 => factor of 1024 in 15 years.
Going forward: Number of cores doubles every 18 months.

Top: Total computing power of top 500 computers
Middle: #1 computer
Bottom: #500 computer

http://www.top500.org



Parallel processing

- Shared memory:
  All processors have access to the same memory. Multicore chip: separate L1 caches, L2 might be shared.
- Distributed memory
  Each processor has it’s own memory and caches. Transferring data between processors is slow. E.g., clusters of computers, supercomputers
- General purpose GPU computing (Graphical Processor Unit)
- Hybrid: Often clusters of multicore/GPU machines!

Multi-thread computing

- For example, multi-threaded program on dual-core computer.
- Thread:
  A thread of control: program code, program counter, call stack, small amount of thread-specific data (registers, L1 cache).
- Shared memory and file system.
  Threads may be spawned and destroyed as computation proceeds.
  Languages like OpenMP.

POSIX Threads

Portable Operating System Interface

Standardized C language threads programming interface
For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.

Multi-thread computing

Some issues:

- Limited to modest number of cores when memory is shared.
- Multiple threads have access to same data — convenient and fast.
- Contention: But, need to make sure they don't conflict (e.g. two threads should not write to same location at same time).
- Dependencies, synchronization: Need to make sure some operations are done in proper order!
- May need cache coherence: If Thread 1 changes x in its private cache, other threads might need to see changed value.

Multi-process computing

A process is a thread that also has its own private address space.

Multiple processes are often running on a single computer
(e.g. different independent programs).

For distributed memory parallel computers, a single computation must be tackled with multiple processes because of memory layout.

Larger cost in creating and destroying processes.
Greater latency in sharing data.

Processes communicate by passing messages.
Languages like MPI — Message Passing Interface.

Multi-process computing with distributed memory

Some issues:

- Often more complicated to program.
- High cost of data communication between processes.
  Want to maximize processing on local data relative to communication with other processes.
- Often need to partition problem domain into subdomains,
  (e.g. domain decomposition for PDEs)
- Generally requires coarse grain parallelism.

Amdahl's Law

Typically only part of a computation can be parallelized.

Suppose 50% of the computation is inherently sequential, and the other 50% can be parallelized.

Question: How much faster could the computation potentially run on many processors?
Answer: At most a factor of 2, no matter how many processors.

The sequential part is taking half the time and that time is still required even if the parallel part is reduced to zero time.

Amdahl’s Law

Suppose 10% of the computation is inherently sequential, and the other 90% can be parallelized.

Question: How much faster could the computation potentially run on many processors?
Answer: At most a factor of 10, no matter how many processors.

The sequential part is taking 1/10 of the time and that time is still required even if the parallel part is reduced to zero time.

Amdahl's Law

Suppose $1/S$ of the computation is inherently sequential,
and the other $(1 - 1/S)$ can be parallelized.

Then can gain at most a factor of S, no matter how many processors.

If T_S is the time required on a sequential machine and we run on P processors, then the time required will be (at least):

$$T_P = (1/S)T_S + (1 - 1/S)T_S/P$$

Note that

$$T_P \to (1/S)T_S \text{ as } P \to \infty$$

Amdahl's Law

Suppose *1/S* of the computation is inherently sequential

$$\Rightarrow T_P = (1/S)T_S + (1 - 1/S)T_S/P$$

Example: If 5% of the computation is inherently sequential (S = 20), then the reduction in time is:

P	T_P
1	T_S
2	$0.525T_S$
4	$0.288T_S$
32	$0.080T_S$
128	$0.057T_S$
1024	$0.051T_S$

Speedup

The ratio T_S/T_P of time on a sequential machine to time running in parallel
is the speedup.

This is generally less than P for P processors. Perhaps much less.

Amdahl's Law plus overhead costs of starting processes/threads, communication, etc.

Caveat: May (rarely) see speedup greater than P ...
For example, if data doesn't all fit in one cache
but does fit in the combined caches of multiple processors.

Scaling

Some algorithms scale better than others as the number of processors increases.

Typically interested on how well algorithms work for large problems requiring lots of time, e.g.

- Particle methods for n particles,
- algorithms for solving systems of n equations,
- algorithms for solving PDEs on $n \times n \times n$ grid in 3D,

For large n, there may be lots of inherent parallelism.

But this depends on many factors:

- dependencies between calculations,
- communication as well as flops,
- nature of problem and algorithm chosen.

Scaling

Typically interested on how well algorithms work for large problems requiring lots of time.

Strong scaling: How does the algorithm perform as the number of processors P increases for a fixed problem size n?
Any algorithm will eventually break down (consider $P > n$)

Weak scaling: How does the algorithm perform when the problem size increases with the number of processors?
E.g. If we double the number of processors can we solve a problem “twice as large” in the same time?

Weak scaling

What does “twice as large” mean?

Depends on how algorithm complexity scales with n.

Example: Solving $n \times n$ linear system with Gaussian elimination requires $O(n^3)$ flops.

Doubling n requires 8 times as many operations.

Problem is “twice as large” if we increase n by a factor of

$2^{1/3} \simeq 1.26$, e.g. from 100 × 100 to 126 × 126.

(Or may be better to count memory accesses!)

Weak scaling

Solving steady state heat equation on $n \times n \times n$ grid.
n^3 grid points =⇒ linear system with this many unknowns.
If we used Gaussian elimination (very bad idea!) we would require ~ $(n^3)^3 = n_9$ flops.
Doubling n would require $2^9 = 512$ times more flops.

Good iterative methods can do the job in $O(n^3)\ log_2(n)$ work or less. (e.g. multigrid).

Developing better algorithms is as important as better hardware!!