Outline

Organization issues

Last week summary

Parallel machines and programming models

MPI Intro
Organization issues

- If you haven’t created an XSEDE account (for using Stampede) and emailed my your user name, please do so as soon as possible.

If you sent me your username before today and haven’t gotten an email from XSEDE, please let me know.

- Pitch your final project to me.

- Thanks for replying to the poll. I assume everybody can compile and run the MPI/OpenMP examples (?)

- Expect a new homework assignment during the week.

- Argonne Training Program on Extreme-scale Computing; application deadline is April 3, 2015.
Outline

Organization issues

Last week summary

Parallel machines and programming models

MPI Intro
Memory hierarchies

Many applications are memory-bounds: the limiting factor is how fast data can be moved to and from the compute unit.

- Memory hierarchies (L1/L2/L3 Cache, DRAM, disc...) to speed up memory access
- Spatial and temporal locality of data access is important.
Memory hierarchies

Many applications are memory-bounds: the limiting factor is how fast data can be moved to and from the compute unit.

- Memory hierarchies (L1/L2/L3 Cache, DRAM, disc...) to speed up memory access
- Spatial and temporal locality of data access is important.

Simple performance model: Only count slow memory access (DRAM) and floating point operations:

\[
\frac{\text{#flop}}{\text{#slow memory access}} = \text{computational intensity.}
\]

Find/develop algorithms with high computational intensity.
Parallelism/Amdahl’s law

Parallelism is crucial and exists on several levels:

- bit level parallelism; pipelining; multiple functional units; (these are taken care of by the compiler, but often human interaction is still necessary)
- process/thread level parallelism: independent processor cores, multicore processors; parallel control flow
- load balance: split work into equal-size parts to avoid idle processes
- Amdahl’s law: If $s$ is the fraction of work one in serial, the maximum speed up of parallelization is $1/s$. 
Parallel scalability
Strong and weak scaling/speedup

Strong scalability

work

cputime

Parallel scalability
Strong and weak scaling/speedup

Strong scalability
Parallel scalability
Strong and weak scaling/speedup

Strong scalability

work

cputime
Parallel scalability
Strong and weak scaling/speedup

Strong scalability
Parallel scalability

Strong and weak scaling/speedup

Strong scalability

Weak scalability

![Graph showing strong scalability](image)

![Graph showing weak scalability](image)
Parallel scalability

Strong and weak scaling/speedup

Strong scalability

Weak scalability

work

cputime

cputime

no of procs

time = 1/efficiency

no of procs
Parallel scalability
Strong and weak scaling/speedup

Strong scalability

Weak scalability

![Diagram showing strong scalability](image)

![Diagram showing weak scalability](image)

Graph showing the relationship between number of processes and efficiency. The x-axis represents the number of processes (1, 2, 4, 8, 16, 32), and the y-axis represents the time (1/efficiency) in percentage.
Parallel scalability

Strong and weak scaling/speedup

Strong scalability

Weak scalability

cputime

cputime

work

work

no of procs

no of procs

time = 1/efficiency

efficiency
Outline

Organization issues

Last week summary

Parallel machines and programming models

MPI Intro
Parallel architectures (Flynn’s taxonomy)

Characterization of architectures according to Flynn:

**SISD:** Single instruction, single data. This is the conventional sequential model.

**SIMD:** Single instruction, multiple data. Multiple processing units with identical instructions, each one working on different data. Useful when a lot of completely identical tasks are needed.

**MIMD:** Multiple instructions, multiple data. Multiple processing units with separate (but often similar) instructions and data/memory access (shared or distributed). We will mainly use this approach.

**MISD:** Multiple instructions, single data. Not practical.
Programming model must reflect architecture

Example: Inner product between two (very long) vectors: $a^T b$:

- Where do $a$, $b$ live? Single memory or distributed?
- What work should be done by which processor?
- How do they coordinate their result?
Shared memory programming model

- Program is a collection of control threads, that are created dynamically
- Each thread has private and shared variables
- Threads can exchange data by reading/writing shared variables
- Danger: more than 1 processor core reads/writes to a memory location: race condition
Shared memory programming model

- Program is a collection of control threads, that are created dynamically
- Each thread has private and shared variables
- Threads can exchange data by reading/writing shared variables
- **Danger**: more than 1 processor core reads/writes to a memory location: race condition

Programming model must manage different threads and avoid race conditions.

**OpenMP**: Open Multi-Processing is the application interface (API) that supports shared memory parallelism: [www.openmp.org](http://www.openmp.org)
Shared memory programming model

Advantages and disadvantages:

+ Relative easy to parallelize loops etc. in serial programs

- Limited amount of parallelism possible in practice

- Memory bus becomes bottleneck if too many processors access the same memory (usually used with $\leq 50$ cores)

- Cache coherency: Need to make sure that values stored in cache of each processor coincide (handled by hardware)

- Size of shared memory is limited and can get very expensive
Distributed memory programming model

- Program is run by a collection of named processes; fixed at start-up
- Local address space; no shared data
- Logically shared data is distributed (e.g., every processor only has direct access to a chunk of rows of a matrix)
- Explicit communication through send/receive pairs
Distributed memory programming model

- Program is run by a collection of named processes; fixed at start-up
- Local address space; no shared data
- Logically shared data is distributed (e.g., every processor only has direct access to a chunk of rows of a matrix)
- **Explicit communication** through send/receive pairs

Programming model must accommodate communication.

Distributed memory programming model

Has become the standard for applications development in HPC: Most large-scale applications are based on MPI

- proved to scale to millions of cores
  - explicit message passing can be technical and creates overhead
  - more intrusive; can require rethinking of algorithms
  - longer code development process; harder to “parallelize” serial code

We’ll start with this programming model.
Hybrid distributed/shared programming model

- Pure MPI approach splits the memory of a multicore processor into independent memory pieces, and uses MPI to exchange information between them.
- **Hybrid approach** uses MPI across processors, and OpenMP for processor cores that have access to the same memory.
- A similar hybrid approach is also used for hybrid architectures, i.e., computers that contain CPUs and accelerators (GPGPUs, MICs).
Other parallel programming approaches

- Grid computing: loosely coupled problems, most famous example was SETI@Home.
- MapReduce: Introduced by Google; targets large data sets with parallel, distributed algorithms on a cluster.
- WebCL
- Pthreads
- CUDA
- Cilk
- ...
Outline

Organization issues

Last week summary

Parallel machines and programming models

MPI Intro
Introduction to MPI

Use B. Gropp’s PPT slides

https://github.com/NYU-HPC15/lecture3