Introduction to Parallel Programming

Linda Woodard
woodard@cac.cornell.edu
October 23, 2013
What is Parallel Programming?

• Theoretically a very simple concept
  – Use more than one processor to complete a task

• Operationally much more difficult to achieve
  – Tasks must be independent
    • Order of execution can’t matter
  – How to define the tasks
    • Each processor works on their section of the problem (functional parallelism)
    • Each processor works on their section of the data (data parallelism)
  – How and when can the processors exchange information
Why Do Parallel Programming?

• Limits of single CPU computing
  – performance
  – available memory

• Parallel computing allows one to:
  – solve problems that don’t fit on a single CPU
  – solve problems that can’t be solved in a reasonable time

• We can solve…
  – larger problems
  – faster
  – more cases
Terminology

• **node**: a discrete unit of a computer system that typically runs its own instance of the operating system
  – Stampede has 6400 nodes

• **processor**: chip that shares a common memory and local disk
  – Stampede has two Sandy Bridge processors per node

• **core**: a processing unit on a computer chip able to support a thread of execution
  – Stampede has 8 cores per processor or 16 cores per node

• **coprocessor**: a lightweight processor
  – Stampede has a one Phi coprocessor per node with 61 cores per coprocessor

• **cluster**: a collection of nodes that function as a single resource
Functional Parallelism

Definition: each process performs a different "function" or executes different code sections that are independent.

Examples:
- 2 brothers do yard work (1 edges & 1 mows)
- 8 farmers build a barn

• Commonly programmed with message-passing libraries
Data Parallelism

Definition: each process does the same work on unique and independent pieces of data

Examples:
- 2 brothers mow the lawn
- 8 farmers paint a barn

- Usually more scalable than functional parallelism
- Can be programmed at a high level with OpenMP, or at a lower level using a message-passing library like MPI or with hybrid programming.
Task Parallelism
a special case of Data Parallelism

Definition: each process performs the same functions but do not communicate with each other, only with a “Master” Process. These are often called “Embarrassingly Parallel” codes.

Examples:
- Independent Monte Carlo Simulations
- ATM Transactions

Stampede has a special wrapper for submitting this type of job; see
https://www.xsede.org/news/-/news/item/5778
Is it worth it to go Parallel?

• Writing effective parallel applications is difficult!!
  – Load balancing is critical
  – Communication can limit parallel efficiency
  – Serial time can dominate

• Is it worth your time to rewrite your application?
  – Do the CPU requirements justify parallelization? Is your problem really “large”?
  – Is there a library that does what you need (parallel FFT, linear system solving)
  – Will the code be used more than once?

10/21/2013  www.cac.cornell.edu
Theoretical Upper Limits to Performance

• All parallel programs contain:
  – parallel sections (we hope!)
  – serial sections (unfortunately)

• Serial sections limit the parallel effectiveness

<table>
<thead>
<tr>
<th></th>
<th>Serial Portion</th>
<th>Parallel Portion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 task</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 tasks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 tasks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Amdahl’s Law states this formally
Amdahl’s Law

• Amdahl’s Law places a limit on the speedup gained by using multiple processors.
  
  – Effect of multiple processors on run time
    
    \[ t_n = \left( \frac{f_p}{N} + f_s \right) t_1 \]
  
  – where
    
    - \( f_s \) = serial fraction of the code
    - \( f_p \) = parallel fraction of the code
    - \( N \) = number of processors
    - \( t_1 \) = time to run on one processor

• Speed up formula: \( S = \frac{1}{f_s + f_p / N} \)
  
  – if \( f_s = 0 \) & \( f_p = 1 \), then \( S = N \)

  – If \( N \to \text{infinity} \): \( S = 1/f_s \); if 10% of the code is sequential, you will never speed up by more than 10, no matter the number of processors.
Practical Limits: Amdahl’s Law vs. Reality

- Amdahl’s Law shows a theoretical upper limit for speedup
- In reality, the situation is even worse than predicted by Amdahl’s Law due to:
  - Load balancing (waiting)
  - Scheduling (shared processors or memory)
  - Communications
  - I/O

![Graph showing speedup vs. number of processors for Amdahl’s Law and Reality with f_p = 0.99](graph.png)
High Performance Computing Architectures
HPC Systems Continue to Evolve Over Time...

Centralized Big-Iron

Decentralized collections


Hybrid Clusters

Mainframes

Mini Computers
Specialized Parallel Computers

RISC Workstations

RISC MPPS

Clusters

NOWS

Grids + Clusters
Cluster Computing Environment

- Login Nodes
- File servers & Scratch Space
- Compute Nodes
- Batch Schedulers
Types of Parallel Computers (Memory Model)

• Useful to classify modern parallel computers by their memory model
  – shared memory
    multiple cores with access to the same physical memory
  – distributed memory
    each task has its own virtual address space
  – hybrid
    mixture of shared and distributed memory; shared memory on cores in a single
    node and distributed memory between nodes

• Most parallel machines today are multiple instruction, multiple data (MIMD)
Shared memory: single address space. All processors have access to a pool of shared memory; easy to build and program, good price-performance for small numbers of processors; predictable performance due to uniform memory access (UMA).

Methods of memory access:
- Bus
- Crossbar

Distributed memory: each processor has its own local memory. Must do message passing to exchange data between processors. cc-NUMA enables larger number of processors and shared memory address space than SMPs; still easy to program, but harder and more expensive to build. (example: Clusters)

Methods of memory access:
- various topological interconnects
Programming Parallel Computers

• Programming single-processor systems is (relatively) easy because they have a single thread of execution and a single address space.

• Programming shared memory systems can benefit from the single address space

• Programming distributed memory systems is more difficult due to multiple address spaces and the need to access remote data

• Programming hybrid memory systems is even more difficult, but gives the programmer much greater flexibility
Single Program, Multiple Data (SPMD)

SPMD:

- One source code is written

- Code can have conditional execution based on which processor is executing the copy

- All copies of code are started simultaneously and communicate and sync with each other periodically
SPMD Programming Model

source.c

source.c
Processor 0

source.c
Processor 1

source.c
Processor 2

source.c
Processor 3
Shared Memory Programming: OpenMP

- Shared memory systems have a single address space:
  - applications can be developed in which loop iterations (with no dependencies) are executed by different processors
  - shared memory codes are mostly data parallel, ‘SIMD’ kinds of codes
  - OpenMP is the new standard for shared memory programming (compiler directives)
  - Vendors offer native compiler directives
Distributed Memory Programming: MPI

Distributed memory systems have separate address spaces for each processor

- Local memory accessed faster than remote memory

- Data must be manually decomposed

- MPI is the standard for distributed memory programming (library of subprogram calls)
Hybrid Memory Programming:

• Systems with multiple shared memory nodes

• Memory is shared at the node level, distributed above that:
  – Applications can be written using OpenMP
  – Applications can be written using MPI
  – Application can be written using both OpenMP and MPI
Questions?