COMP 322: Fundamentals of Parallel Programming

Lecture 1:
The What and Why of Parallel Programming; Task Creation & Termination (async, finish)

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu
https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Acknowledgments

—CS 194 course on “Parallel Programming for Multicore” taught by Prof. Kathy Yelick, UC Berkeley, Fall 2007
  - http://www.cs.berkeley.edu/~yelick/cs194f07/
—“Principles of Parallel Programming”, Calvin Lin & Lawrence Snyder, Addison-Wesley 2009
—COMP 322 Module 1 handout, Sections 0.1, 0.2, 1.1
—edX lecture and demonstration videos for Module 1, topic 1.1
What is Parallel Computing?

- **Parallel computing**: using multiple processors in parallel to solve problems more quickly than with a single processor and/or with less energy.

- Examples of a parallel computer:
  - An 8-core Symmetric Multi-Processor (SMP) consisting of four dual-core Chip Multi-Processors (CMPs).

Source: Figure 1.5 of Lin & Snyder book, Addison-Wesley, 2009
All Computers are Parallel Computers --- Why?
Moore’s Law

Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 1–2 years.

Resulted in CPU clock speed doubling roughly every 18 months, but not any longer.

Slide source: Jack Dongarra
Current Technology Trends

- Chip density is continuing to increase ~2x every 2 years
  - Clock speed is not
  - Number of processors is doubling instead

- Parallelism must be managed by software

Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)
Parallelism Saves Power
(Simplified Analysis)

Power = (Capacitance) * (Voltage)^2 * (Frequency)

⇒ Power is proportional to (Frequency)^3

Baseline example: single 1GHz core with power P

Option A: Increase clock frequency to 2GHz ⇒ Power = 8P

Option B: Use 2 cores at 1 GHz each ⇒ Power = 2P

• Option B delivers same performance as Option A with 4x less power ... provided software can be decomposed to run in parallel!
A Real World Example

- Fermi vs. Kepler GPU chips from NVIDIA's GeForce 600 Series

  Source: [http://www.theregister.co.uk/2012/05/15/nvidia_kepler_tesla_gpu_revealed/](http://www.theregister.co.uk/2012/05/15/nvidia_kepler_tesla_gpu_revealed/)

<table>
<thead>
<tr>
<th></th>
<th>Fermi chip (released in 2010)</th>
<th>Kepler chip (released in 2012)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>512</td>
<td>1,536</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>1.3 GHz</td>
<td>1.0 GHz</td>
</tr>
<tr>
<td>Power</td>
<td>250 Watts</td>
<td>195 Watts</td>
</tr>
<tr>
<td>Peak double precision</td>
<td>665 Gigaflops</td>
<td>1310 Gigaflops (1.31 Teraflops)</td>
</tr>
<tr>
<td>floating point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>performance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scope of Course

• Fundamentals of parallel programming
  — Primitive constructs for task creation & termination, collective & point-to-point synchronization, task and data distribution, and data parallelism
  — Abstract models of parallel computations and computation graphs
  — Parallel algorithms & data structures including lists, trees, graphs, matrices
  — Common parallel programming patterns
• Habanero-Java (HJ) library as a pedagogic parallel programming model
  — Developed in the Habanero Multicore Software Research project at Rice
• Java Concurrency
• Beyond HJ and Java: Map-Reduce, CUDA, MPI
• Weekly quizzes in edX (due by Friday for each week)
• 6 Homeworks with written assignments and programming assignments
  — Abstract metrics
  — Real parallel systems in Rice’s Data Center
What is Parallel Programming?

- Specification of operations that can be executed in parallel
- A parallel program is decomposed into sequential subcomputations called tasks
- Parallel programming constructs define task creation, termination, and interaction
Example of a Sequential Program: Computing the sum of array elements

```java
int sum = 0;
for (int i=0 ; i < X.length ; i++)
    sum += X[i];
```

Observations:

• The decision to sum up the elements from left to right was arbitrary

• The computation graph shows that all operations must be executed sequentially
Parallelization Strategy for two cores  
(Two-way Parallel Array Sum)

Basic idea:

- Decompose problem into two tasks for partial sums
- Combine results to obtain final answer
- Parallel divide-and-conquer pattern

Task 0: Compute sum of lower half of array

Task 1: Compute sum of upper half of array

Compute total sum
Async and Finish Statements for Task Creation and Termination

<table>
<thead>
<tr>
<th>async S</th>
<th>finish S</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Creates a new child task that executes statement S</td>
<td>• Execute S, but wait until all asyncs in S’s scope have terminated.</td>
</tr>
</tbody>
</table>

```
// T0 (Parent task)
STMT0;
finish {  //Begin finish
    async {
        STMT1;  //T1 (Child task)
    }
    STMT2;  //Continue in T0
    //Wait for T1
}  //End finish
STMT3;  //Continue in T0
```
Two-way Parallel Array Sum using async & finish constructs

1. // Start of Task T0 (main program)
2. sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
3. finish {
4. async // Child task computes sum of lower half of array
5. for(int i=0; i < X.length/2; i++) sum1 += X[i];
6. // Parent task computes sum of upper half of array
7. for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
8. }
9. // Parent task waits for child task to complete (join)
10. return sum1 + sum2;

NOTE: This example uses pseudocode notation for async & finish instead of concrete Habanero Java library syntax
Lecture Modules

1. **Deterministic Shared-Memory Parallelism**: creation and coordination of parallelism, collective & point-to-point synchronization (phasers, barriers), abstract performance metrics (work, span, critical paths), Amdahl's Law, weak vs. strong scaling, data races and determinism, data race avoidance (immutability, futures, accumulators, dataflow), deadlock avoidance, abstract vs. real performance (granularity, scalability), parallel sorting algorithms.

2. **Nondeterministic Shared-Memory Parallelism and Concurrency**: critical sections, atomicity, isolation, high level data races, nondeterminism, linearizability, liveness/progress guarantees, actors, request-response parallelism

3. **Distributed-Memory Parallelism and Locality**: memory hierarchies, cache affinity, false sharing, message-passing (MPI), communication overheads (bandwidth, latency), MapReduce, systolic arrays, accelerators, GPGPUs.
COMP 322 Course Information: Spring 2014

• IMPORTANT:
  — Send email to comp322-staff@mailman.rice.edu if you did NOT receive a welcome email from us
  — Bring your laptop to this week’s lab (Section A01: Monday, Section A02: Wednesday) to ensure that you are properly set up with all class infrastructure

• Course Requirements:
  — Homeworks (6) 40% (written + programming components)
  — Exams (2) 40% (take-home written exams)
  — Weekly Quizzes 10% (on EdX)
  — Class Participation 10% (in-class activities, lab attendance, classroom & on-line discussions, Q&A)

• HW1 will be assigned on Jan 17th and be due on Jan 31st