COMP 322: Fundamentals of Parallel Programming

Lecture 1: The What and Why of Parallel Programming

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
COMP 322 Course Information: Spring 2013

• “Fundamentals of Parallel Programming”

• Lectures: MWF, 1pm - 1:50pm, Hertzstein 212

• Labs, Ryon 102 (mandatory):
  — Tuesdays, 4:00pm - 5:20pm (section A03)
  — Wednesdays, 3:30pm - 4:50pm (section A02)
  — Thursdays, 4:00pm - 5:20pm (section A01)

• Instructor: Vivek Sarkar (vsarkar@rice.edu)

• TAs: Kumud Bhandari, Max Grossman, Deepak Majeti, Annirudh Prasad, Rishi Surendran, Yunming Zhang

• Prerequisite: COMP 215 or equivalent

• Cross-listing: ELEC 323
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) as a pedagogic language, developed in the Habanero Multicore Software Research project at Rice

• Java Concurrency

• Beyond HJ and Java: Map-Reduce, CUDA, MPI

• Weekly Lab and Lecture Quizzes

• Homeworks with written assignments and programming assignments
  — Abstract metrics
  — Real parallel systems (8-core Intel, Rice SUG@R system)
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.

4. **Current Practice — today's Parallel Programming Models and Challenges**: Java Concurrency, locks, condition variables, semaphores, memory consistency models, comparison of parallel programming models (.Net Task Parallel Library, OpenMP, CUDA, OpenCL); energy efficiency, data movement, resilience.
Outline of Today’s Lecture

• Introduction

• Async-Finish Parallel Programming

• 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 1.1, 1.2, 2.1, 2.2
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
Number of processors in the world’s fastest computers during 2005-2011

Source: http://www.top500.org
All Computers are Parallel Computers --- Why?
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)

\[
\text{Power} = (\text{Capacitance}) \times (\text{Voltage})^2 \times (\text{Frequency})^3
\]

\[\Rightarrow\] Power is proportional to \((\text{Frequency})^3\)

**Baseline example:** single 1GHz core with power \(P\)

**Option A:** Increase clock frequency to 2GHz \[\Rightarrow\] Power = 8\(P\)

**Option B:** Use 2 cores at 1 GHz each \[\Rightarrow\] Power = 2\(P\)

• **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 performance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

Schematic of a dual-core Processor
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)

Task 0: Compute sum of lower half of array

Task 1: Compute sum of upper half of array

Compute total sum

Basic idea:

• Decompose problem into two tasks for partial sums
• Combine results to obtain final answer
• Parallel divide-and-conquer pattern
Outline of Today’s Lecture

• Introduction
• Async-Finish Parallel Programming
Async and Finish Statements for Task Creation and Termination

**async S**
- Creates a new child task that executes statement S

**finish S**
- Execute S, but wait until all asyncs in S’s scope have terminated.

// T₀ (Parent task)
STMT0;
finish {
  async {
    STMT1; //T₁ (Child task)
  }
  STMT2; //Continue in T₀
  //Wait for T₁
}
//End finish
STMT3; //Continue in T₀

T₁
T₀
STMT0

---

fork

T₁
T₀

---

join

STMT1
STMT2

---

STMT3
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. async { // Task T1 computes sum of upper half of array
4.     for (int i=X.length/2; i < X.length; i++)
5.         sum2 += X[i];
6. }
7. // T0 computes sum of lower half of array
8. for (int i=0; i < X.length/2; i++) sum1 += X[i];
9. // Task T0 waits for Task T1 (join)
10. return sum1 + sum2;

Where does finish go?
Time for worksheet #1!
Some Properties of Async & Finish constructs

1. Scope of async/finish can be any arbitrary statement
   - async/finish constructs can be arbitrarily nested e.g.,
   - finish { async S1; finish { async S2; S3; } S4; } S5;

2. A method may return before all its async's have terminated
   - Enclose method body in a finish if you don't want this to happen
   - main() method is enclosed in an implicit finish e.g.,
   - main(){ foo();} void foo() {async S1; S2; return;}

3. Each dynamic async task will have a unique Immediately Enclosing Finish (IEF) at runtime

4. Async/finish constructs cannot "deadlock"
   - Cannot have a situation where both task A waits for task B to finish, and task B waits for task A to finish

5. Async tasks can read/write shared data via objects and arrays
   - Local variables have special restrictions
COMP 322 Course Information: Spring 2013

• IMPORTANT:
  1. Send email to comp322-staff@mailman.rice.edu if you did NOT receive a welcome email from us
  2. Apply for a Coursera account and send the email address for your account to comp322-staff@mailman.rice.edu
     • We will use Coursera for on-line quizzes & discussions

• Course Requirements:
  —Homeworks (6) 40% (written + programming components)
  —Exams (2) 40% (take-home written exams)
  —Weekly Quizzes 10% (one lecture + one lab quiz per week)
  —Class Participation 10% (lecture worksheets, lab attendance, classroom & on-line discussions, Q&A)

• HW1 will be assigned on Jan 9th and be due on Jan 23rd
Worksheet #1 (to be done individually or in pairs): Insert finish to get correct Two-way Parallel Array Sum program

Name 1: ___________________          Name 2: ___________________

1.  // Start of Task T0 (main program)
2. sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
3. async { // Task T1 computes sum of upper half of array
4.   for(int i=X.length/2; i < X.length; i++)
5.       sum2 += X[i];
6. }
7.  // T0 computes sum of lower half of array
8. for(int i=0; i < X.length/2; i++) sum1 += X[i];
9.  // Task T0 waits for Task T1 (join)
10. return sum1 + sum2;