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
Scope of Course

• Fundamentals of parallel programming
  — Task creation and termination, computation graphs, scheduling theory, futures, forall parallel loops, barrier synchronization (phasers), isolation & mutual exclusion, task affinity, bounded buffers, data flow, threads, GUI applications, data races, deadlock, memory models

• Introduction to parallel algorithms

• Habanero-Java (HJ) language, developed in the Habanero Multicore Software Research project at Rice

• Abstract executable performance model for HJ programs

• Java Concurrency

• Written assignments

• Programming assignments
  — Abstract metrics
  — Real parallel systems (8-core Intel, Rice SUG@R system)

• Beyond HJ and Java: introduction to CUDA and MPI
Acknowledgments for Today’s Lecture

• CS 194 course on “Parallel Programming for Multicore” taught by Prof. Kathy Yelick, UC Berkeley, Fall 2007
  —http://www.cs.berkeley.edu/~yelick/cs194f07/

• COMP 322 Lecture 1 handout
What is Parallel Computing?

• **Parallel computing:** using multiple processors in parallel to solve problems more quickly than with a single processor, or with less energy

• Examples of parallel machines
  — A computer Cluster that contains multiple PCs with local memories combined together with a high speed network
  — A Symmetric Multi-Processor (SMP) that contains multiple processor chips connected to a single shared memory system
  — A Chip Multi-Processor (CMP) contains multiple processors (called cores) on a single chip, also called Multi-Core Computers

• The main motivation for parallel execution historically came from the desire for improved performance
  — Computation is the third pillar of scientific endeavor, in addition to Theory and Experimentation

• But parallel execution has also now become a ubiquitous necessity due to power constraints, as we will see
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

Task A

Task B
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
Async and Finish Statements for Task Creation and Termination

**async S**
- Creates a new child task that executes statement S
- Parent task immediately continues to statement following the async

```
//Task T₀(Parent)
finish { //Begin finish
    async
        STMT₁; //T₁(Child)
    //Continuation
    STMT₂; //T₀
} //Continuation //End finish
STMT₃; //T₀
```

**finish S**
- Execute S, but wait until all (transitively) spawned asyncs in S’s scope have terminated.
- Implicit finish between start and end of main program

```
T₁
async
    STMT₁; //T₁(Child)
    //Continuation
    STMT₂; //T₀
} //Continuation //End finish
T₀

T₀
STMT₃
```

**Tasks and Terminations**
- **async**: Creates a new child task that executes the given statement.
- **finish**: waits for all transitively spawned asyncs in the given scope to terminate before continuing.

**Example Diagram**
- `async STMT₁`: Creates a new child task `T₁`.
- `async STMT₂`: Creates another child task `T₂`.
- `finish { async STMT₁; }`: Waits for all transitively spawned asyncs in `STMT₁`’s scope to terminate.
- `T₀`: Continuation of the main program.
- `T₁`: Continuation of the child task `T₁`.

**Termination Phases**
- **Wait**: Waits for all transitively spawned asyncs to terminate.
- **Terminate**: Ends the task and releases any resources.

**Control Flow**
- `STMT₁`: Executed by `T₁`.
- `STMT₂`: Executed by `T₀`.
- `STMT₃`: Continuation of the main program after completion.

**Observation**
- Task `T₁` is created before `STMT₁` is executed.
- Task `T₀` is created after `STMT₁` is executed.
- Task `T₀` is completed after `STMT₃` is executed.
Example of a Parallel Program: Array Sum with two tasks

// Start of Task T1 (main program)
sum1 = 0; sum2 = 0;
// Assume that sum1 & sum2 are fields
finish {
    // Compute sum1 (lower half) and sum2
    // (upper half) in parallel
    async for (int i=0; i < X.length/2; i++)
        sum1 += X[i]; // Task T2
    async for (int i=X.length/2; i < X.length; i++)
        sum2 += X[i]; // Task T3
}
// Task T1 waits for Tasks T2 and T3
int sum = sum1 + sum2; // Continuation of Task T1

Computation Graph

// Start of Task T1 (main program)
Why Parallel Computing Now?

- Researchers have been using parallel computing for decades:
  - Mostly used in computational science and engineering
  - Problems too large to solve on one computer; use 100s or 1000s

- There have been higher level courses in parallel computing (COMP 422, COMP 522) at Rice for several years

- Many companies in the 80s/90s “bet” on parallel computing and failed
  - Sequential computers got faster too quickly for there to be a large market for specialized parallel computers

- Why is Rice adding a 300-level undergraduate course on parallel programming now?
  - Because the entire computing industry has bet on parallelism
  - There is a desperate need for all computer scientists and practitioners to be aware of parallelism
Number of processors used in Top 500 computers from 1993 to 2010

Source: www.top500.org
Technology Trends: Microprocessor Capacity

2X transistors/Chip every 1-2 years

Called "Moore’s Law"

Microprocessors have become smaller, denser, and more powerful.

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

Slide source: Jack Dongarra
Microprocessor Transistors and Clock Rate

**Growth in transistors per chip**

**Increase in clock rate**

Old view: why bother with parallel programming for increased performance? Just wait a year or two...
Power Wall

Scaling clock speed (business as usual) will not work

Source: Patrick Gelsinger, Intel®
Parallelism Saves Power

Power = (Capacitance) \times (Voltage)^2 \times (Frequency)

\Rightarrow \text{Power} \propto (Frequency)^3

Baseline example: single 1GHz core with power P

Option A: Increase clock frequency to 2GHz \Rightarrow \text{Power} = 8P

Option B: Use 2 cores at 1 GHz each \Rightarrow \text{Power} = 2P

• Option B delivers same performance as Option A with 4x less power ... provided software can be decomposed to run in parallel!
Revolution is Happening Now

- Chip density is continuing to increase ~2x every 2 years
  - Clock speed is not
  - Number of processor cores may double instead
- There is little instruction-level parallelism (ILP) to be found by hardware
- Parallelism must be exposed to and managed by software

Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)
Implications

• These arguments are no long theoretical

• All major processor vendors are producing multicore chips
  – Every machine will soon be a parallel machine
  – All programmers will be parallel programmers???

• Some may eventually be hidden in libraries, compilers, and high level languages
  – But a lot of work is needed to get there

• Big open questions:
  – What will be the killer applications for multicore machines?
  – How should the chips be designed?
  – How will they be programmed?