

# Lecture 13—Parallelization Patterns

ECE 459: Programming for Performance

January 30, 2015

# Estimating Impact of Speculative Execution

$T_1$ : time to run longCalculatuion.

$T_2$ : time to run secondLongCalculation.

$p$ : probability that secondLongCalculation executes.

In the normal case we have:

$$T_{\text{normal}} = T_1 + pT_2.$$

$S$ : synchronization overhead.

Our speculative code takes:

$$T_{\text{speculative}} = \max(T_1, T_2) + S.$$

**Exercise.** When is speculative code faster? Slower?  
How could you improve it?

# Shortcomings of Speculative Execution

Consider the following code:

```
void doWork(int x, int y) {  
    int value = longCalculation(x, y);  
    return secondLongCalculation(value);  
}
```

Now we have a true dependency; can't use speculative execution.

But: if the value is predictable, we can execute `secondLongCalculation` using the predicted value.

This is **value speculation**.

# Value Speculation Implementation

This Pthread code does value speculation:

```
void doWork(int x, int y) {  
    thread_t t1, t2;  
    point p(x,y);  
    int v1, v2, last_value;  
    thread_create(&t1, NULL, &longCalculation, &p);  
    thread_create(&t2, NULL, &secondLongCalculation, &last_value);  
    thread_join(t1, &v1);  
    thread_join(t2, &v2);  
    if (v1 == last_value) {  
        return v2;  
    } else {  
        last_value = v1;  
        return secondLongCalculation(v1);  
    }  
}
```

Note: this is like memoization (plus parallelization).

## Estimating Impact of Value Speculation

$T_1$ : time to run `longCalculation`.

$T_2$ : time to run `secondLongCalculation`.

$p$ : probability that `secondLongCalculation` executes.

$S$ : synchronization overhead.

In the normal case, we again have:

$$T = T_1 + pT_2.$$

This speculative code takes:

$$T = \max(T_1, T_2) + S + pT_2.$$

**Exercise.** Again, when is speculative code faster?  
Slower? How could you improve it?

# When Can We Speculate?

Required conditions for safety:

- `longCalculation` and `secondLongCalculation` must not call each other.
- `secondLongCalculation` must not depend on any values set or modified by `longCalculation`.
- The return value of `longCalculation` must be deterministic.

General warning: Consider [side effects](#) of function calls.

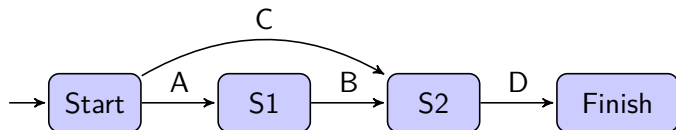
# Part I

## Parallelization Patterns

# Critical Paths

Should be familiar with critical paths from other courses (Gantt charts).

Consider the following diagram (edges are tasks):



- B depends on A, C has no dependencies, and D depends on B and C.
- Can execute A-then-B in parallel with C.
- Keep dependencies in mind when calculating speedups for more complex programs.



# Data and Task Parallelism

**Data parallelism** is performing *the same* operations on different input.

**Example:** doubling all elements of an array.

**Task parallelism** is performing *different* operations on different input.

**Example:** playing a video file: one thread decompresses frames, another renders.

# Data Parallelism: Single Instruction, Multiple Data

We'll discuss SIMD in more detail later. An overview:

- You can load a bunch of data and perform arithmetic.
- Instructions process multiple data items simultaneously.  
(Exact number is hardware-dependent).

For x86-class CPUs, MMX and SSE extensions provide SIMD instructions.

# SIMD Example

Consider the following code:

```
void vadd(double * restrict a, double * restrict b,  
          int count) {  
    for (int i = 0; i < count; i++)  
        a[i] += b[i];  
}
```

In this scenario, we have a regular operation over block data.

We could use threads, but we'll use SIMD.

## SIMD Example—Assembly without SIMD

If we compile this without SIMD instructions on a 32-bit x86, (flags -m32 -march=i386 -S) we might get this:

```
loop:
    fldl    (%edx)
    faddl   (%ecx)
    fstpl   (%edx)
    addl    8, %edx
    addl    8, %ecx
    addl    1, %esi
    cmp     %eax, %esi
    jle     loop
```

Just loads, adds, writes and increments.

## SIMD Example—Assembly with SIMD

Instead, compiling to SIMD instructions  
(`-m32 -mfpmath=sse -march=prescott`) gives:

```
loop :  
    movupd (%edx),%xmm0  
    movupd (%ecx),%xmm1  
    addpd  %xmm1,%xmm0  
    movpd  %xmm0,(%edx)  
    addl   16,%edx  
    addl   16,%ecx  
    addl   2,%esi  
    cmp    %eax,%esi  
    jle    loop
```

- Now processing two elements at a time on the same core.
- Also, no need for stack-based x87 code.

# SIMD Overview

- Operations *packed*: operate on multiple data elements at the same time.
- On modern 64-bit CPUs, SSE has 16 128-bit registers.
- Very good if your data can be *vectorized* and performs math.
- Usual application: image/video processing.
- We'll see more SIMD as we get into GPU programming: GPUs excel at these types of applications.

# Task-Based Patterns: Overview

- We'll now look at thread and process-based parallelization.
- Although threads and processes differ, we don't care for now.

## Pattern 1: Multiple Independent Tasks

Only useful to maximize system utilization.

- Run multiple tasks on the same system (e.g. database and web server).

If one is memory-bound and the other is I/O-bound, for example, you'll get maximum utilization out of your resources.

**Example:** cloud computing, each task is independent and can tasks can spread themselves over different nodes.

- Performance can increase linearly with the number of threads.



## Pattern 2: Multiple Loosely-Coupled Tasks

Tasks aren't quite independent, so there needs to be some inter-task communication (but not much).

- Communication might be from the tasks to a controller or status monitor.

Refactoring an application can help with latency.

For instance: split off the CPU-intensive computations into a separate thread—your application may respond more quickly.

**Example:** A program (1) receives/forwards packets and (2) logs them. You can split these two tasks into two threads, so you can still receive/forward while waiting for disk. This will increase the **throughput** of the system.

## Pattern 3: Multiple Copies of the Same Task

Variant of multiple independent tasks: run multiple copies of the same task (probably on different data).

- No communication between different copies.

Again, performance should increase linearly with number of tasks.

**Example:** In a rendering application, each thread can be responsible for a frame (gain **throughput**; same **latency**).

## Pattern 4: Single Task, Multiple Threads

Classic vision of “parallelization”.

**Example:** Distribute array processing over multiple threads—each thread computes results for a subset of the array.

- Can decrease **latency** (and increase **throughput**), as we saw with **Amdahl's Law**.
- Communication can be a problem, if the data is not nicely partitioned.
- Most common implementation is just creating threads and joining them, combining all results at the join.

## Pattern 5: Pipeline of Tasks

Seen briefly in computer architecture.

- Use multiple stages; each thread handles a stage.

**Example:** a program that handles network packets: (1) accepts packets, (2) processes them, and (3) re-transmits them. Could set up the threads such that each packet goes through the threads.

- Improves **throughput**; may increase **latency** as there's communication between threads.
- In the best case, you'll have a linear speedup.

Rare, since the runtime of the stages will vary, and the slow one will be the bottleneck (but you could have 2 instances of the slow stage).

## Pattern 6: Client-Server

To execute a large computation, the server supplies work to many clients—as many as request it.

Client computes results and returns them to the server.

**Examples:** botnets, SETI@Home, GUI application (backend acts as the server).

Server can arbitrate access to shared resources (such as network access) by storing the requests and sending them out.

- Parallelism is somewhere between single task, multiple threads and multiple loosely-coupled tasks

## Pattern 7: Producer-Consumer

Variant on the pipeline and client-server models.  
Producer generates work, and consumer performs work.

**Example:** producer which generates rendered frames;  
consumer which orders these frames and writes them to  
disk.

Any number of producers and consumers.

- This approach can improve **throughput** and also reduces design complexity

# Combining Strategies

Most problems don't fit into one category, so it's often best to combine strategies.

For instance, you might often start with a pipeline, and then use multiple threads in a particular pipeline stage to handle one piece of data.

Tip: estimate to see what divisions of strategies would work best (might have to do more iterations of Amdahl's law depending on the amount of strategies you can use).

## Midterm Questions from 2011 (1)

For each of the following situations, **name an appropriate parallelization pattern and the granularity at which you would apply it, explain the necessary communication, and explain why your pattern is appropriate.**

- build system, e.g. parallel make
- optical character recognition system



## Midterm Questions from 2011 (1)

For each of the following situations, **name an appropriate parallelization pattern and the granularity at which you would apply it, explain the necessary communication, and explain why your pattern is appropriate.**

- build system, e.g. parallel make
  - ▶ Multiple independent tasks, at a per-file granularity
- optical character recognition system
  - ▶ Pipeline of tasks
  - ▶ 2 tasks - finding characters and analyzing them

## Midterm Questions from 2011 (2)

**Give a concrete example** where you would use the following parallelization patterns. **Explain** the granularity at which you'd apply the pattern.

- single task, multiple threads:
- producer-consumer (no rendering frames, please):

## Midterm Questions from 2011 (2)

**Give a concrete example** where you would use the following parallelization patterns. **Explain** the granularity at which you'd apply the pattern.

- single task, multiple threads:
  - ▶ Computation of a mathematical function with independent sub-formulas.
- producer-consumer (no rendering frames, please):
  - ▶ Processing of stock-market data: a server might generate raw financial data (quotes) for a particular security. The server would be the producer. Several clients (or consumers) may take the raw data and use them in different ways, e.g. by computing means, generating charts, etc.

## Part II

### Parallelizing Code

# How To Parallelize Code: Strategy

Four-step outline:

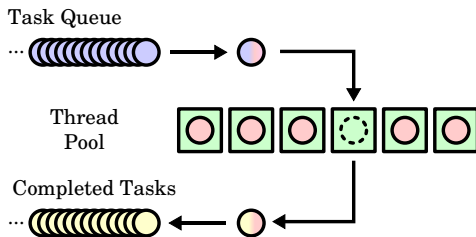
- ① Profile the code.
- ② Look at hotspots; find and optimize dependencies; parallelize dependency chains; change the algorithm if you can.
- ③ Estimate benefits.
- ④ If not good enough, step back and try higher level of abstraction.

Always try to minimize synchronization.

## Low-level Implementation Tactic: Thread Pools

Instead of creating threads, destroying them and recreating them, you can use a **thread pool**.

- It creates  $n$  threads; you just push work onto them.



- Only question is: How many threads should you create? (Experiment to find out).
- Implementation from GLib: `GThreadPool`.

# Introduction to Automatic Parallelization

Vision: take a sequential C program and automatically convert it into a parallel version.

Lots of research in the early 1990s, then tapered off.  
(it's hard!)

Renewed interest now since multicores are so common.  
(it's still hard!)

# What Can We Parallelize?

- Some languages are easier than others to reason about (and therefore to automatically parallelize).
- C can be easy to parallelize, given the right code, plus compiler hints.
- “The right code” = arrays with no loop-carried dependencies.



# Automatic Parallelization in Practice

Some production compilers support automatic parallelization:

- `icc` (Intel's non-free compiler);
- `solarisstudio` (Oracle's free-as-in-beer compiler <sup>1</sup>);
- `gcc` (GNU's free-as-in-speech compiler).

---

<sup>1</sup><http://www.oracle.com/technetwork/documentation/solaris-studio-12-192994.html>

## Example Code from the Textbook

We saw automatic parallelization of some code.  
Let's revisit the whole issue in Lecture 10.