Lecture 07 - Speculation and Parallelization Patterns

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

January 18, 2012

Breaking Dependencies

- Recall that computer architects use speculation to predict branch targets
- This gets around having to wait for the branch to be evaluated until you can continue doing useful work
- We can also use speculation at a coarser-grained level and speculatlively parallelize code
- We'll see two ways: speculative execution and value speculation

Example

Consider the following code:

```
void doWork(int x, int y) {
    int value = longCalculation(x, y);
    if (value > threshold) {
      return value + secondLongCalculation(x, y);
    }
    else {
      return value;
    }
}
```

Obviously, we don't know whether or not we'll have to do `secondLongCalculation`

- Could we execute `longCalculation` and `secondLongCalculation` in parallel if we didn't have the conditional?

Example with Speculative Execution

Well yes, we could, consider this pseudocode:

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

We do both the calculations in parallel and return the same result as before

- When is this code faster? Slower? How could you improve?

Example Formulas for Speculative Execution

Let T_1 be the time to run `longCalculatuion`
Let T_2 be the time to run `secondLongCalculatuion`
Let p be the probability that `secondLongCalculatuion` executes

In the normal case we have:

$$T = T_1 + pT_2$$

Let S be the synchronization overhead

Our speculative code takes:

$$T = max(T_1, T_2) + S$$

Another Example

Consider the following code:

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

Now we have a true dependency and can't do the same strategy as before

- If the value is predictable however, we can execute `secondLongCalculation` based on a predicted value

Example with Value Speculation

Consider this pseudocode:

```
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);
    }
}
```

Similar to memoization (except with parallelization thrown in)

Example Formulas for Value Speculation

Let T_1 be the time to run `longCalculatuion`
Let T_2 be the time to run `secondLongCalculatuion`
Let p be the probability that `secondLongCalculatuion` executes
Let S be the synchronization overhead

In the normal case we have:

$$T = T_1 + pT_2$$

Our speculative code takes:

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

Other Considerations

In order for this to be safe to parallelize we must meet these conditions:

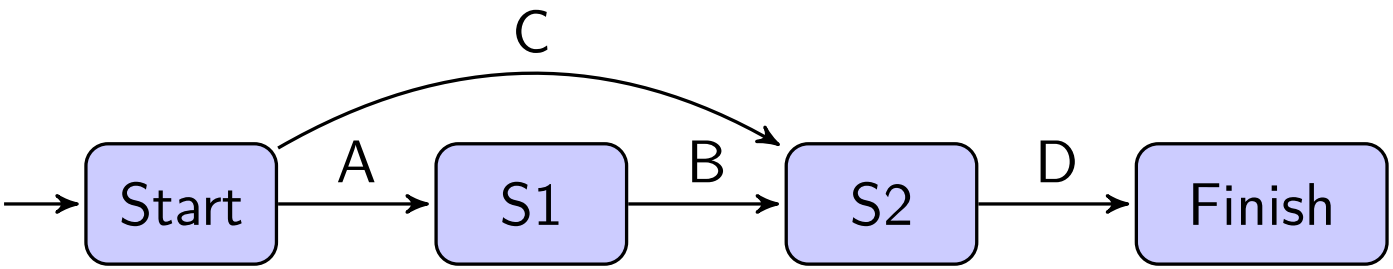
- `longCalculation` and `secondLongCalculation` must not call each other
- The implementation of `secondLongCalculation` must not depend on any values set or modified by `longCalculation`
- The return value of `longCalculation` must be deterministic

- Always consider the side effects of any function calls you make

Critical Paths

- Should be familiar with critcal paths from other course (Gantt charts)

Consider the following diagram:



- B depends on A, C has no dependencies and D depends on B and C
- You can execute A and B in parallel with C
- Should always have this 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 separate input
- **Example:** playing a video file, one thread for decompressing frames and another for rendering

Single Instruction, Multiple Data

- We'll focus more on SIMD later in the course, but it's good to know
- Intructions, obviously, work on a bunch of data simultaneously (the exact number is hardware dependent)
- Data is understood in blocks, so you can load a bunch and perform some arithmetic
- For x86 class CPUs, these instructions are provided from MMX and SSE

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 the same operation over block data
- We could divide this up as well using threads, but can also use SIMD

SIMD Example - Assembly without SIMD

If we compile this without SIMD instructions on an x86, 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
```

- Simply just loads, adds, writes and increments

SIMD Example - Assembly with SIMD

We can instead compile to SIMD instructions and get something like this:

```
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
```

- We're doing two elements at a time now, on the same core

SIMD Overview

- Operations are *packed* and operate on multiple data elements at the same time
- For 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 about SIMD as we get into GPU programming (they're very good at these types of applications)

Overview

- In the following examples we'll be looking at thread or process-based parallelization

- Again, we should be familiar with differences between a thread and process

Multiple Independent Tasks

- Only useful to maximize system utilization
- Running multiple tasks on the same system (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 spread itself over different nodes
- Performance should increase linearly with the number of threads

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, e.g. splitting off the CPU-intensive computations into a thread, then your application may respond more quickly
- **Example:** A program receives/forward packets and logs them. You can split these two tasks into two threads, so you can still recieve/foward while waiting for disk. This will increase the throughput of the system

Multiple Copies of the Same Task

- Variant of multiple independent tasks
- Run multiple copies of the same task (probably on different data)
- In this case, there would be no communcation 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)

Single Task, Multiple Threads

- Classic vision of "parallelization"
- **Example:** Distributing array processing over multiple threads (each thread computes results for a subset of the array)
- Can decrease latency (also increases throughput) as we saw with Amdahl's Law
- Communication can be a problem, if the data is not nicely separated
- Most common implementation is just creating threads and joining them, combining all the results at the join

Pipeline of Tasks

- Seen this briefly in computer architecture
- Have multiple stages, with each thread doing a stage
- **Example:** a program that handles network packets, it: accepts packets, processes them and re-transmits them. Could set up the threads where a packet goes through the threads.
- Improve throughput, may increase latency since there's communication between threads
- In the best case, you'll have a linear speedup
- Rare since the runtime of the stages will not be even, and the slow one will be the bottleneck (although you could have 2 instances of the slow stage)

Client-Server

- To execute a large computation the server supplies work to many clients (as many as request it)
- Client computes the results and returns the result 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

Producer-Consumer

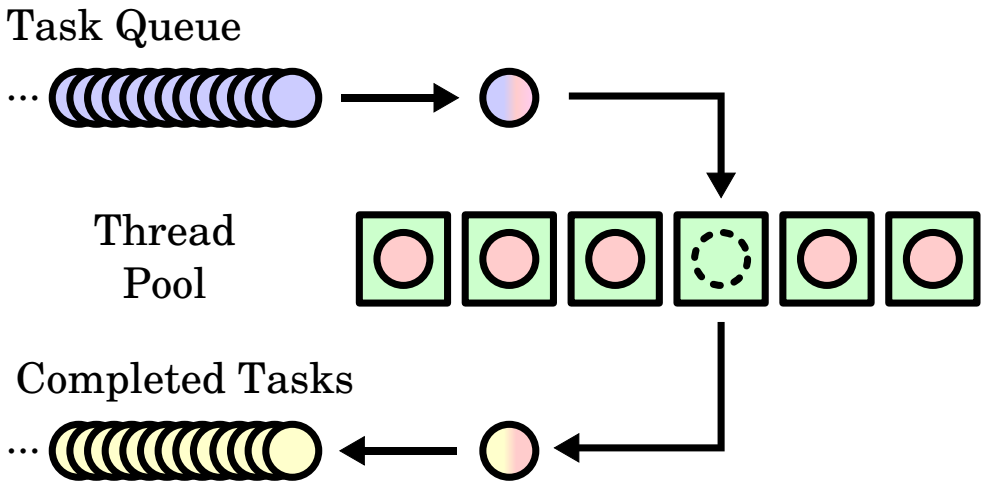
- Variant on the pipeline and client-server models
- Producer generates work, and the consumer performs work
- **Example:** a producer which generates rendered frames, and a consumer which orders these frames and writes them to disk
- Any number of producers and consumers
- This approach can improve throughput and also reduces design complexity

Combining Strategies

- Most problems don't fit into one category, 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
- Always 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)

Thread Pools

- Instead of creating threads, destroying them and recreating them, you can use a thread pool
- Creates a n threads and you just `push` work onto them



- Only question is, how many threads should you create? (you should have a pretty good feel after Assignment 1)
- Common implementation: `GThreadPool`

Midterm Questions from Last Year (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 Last Year (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 Last Year (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 Last Year (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. For instance by generating averages, means, charts, etc.