Parallel Programming

Recitation Session 5

Thomas Weibel <weibelt@ethz.ch>

Laboratory for Software Technology,
Swiss Federal Institute of Technology Zürich

April 1, 2010

Executive Summary

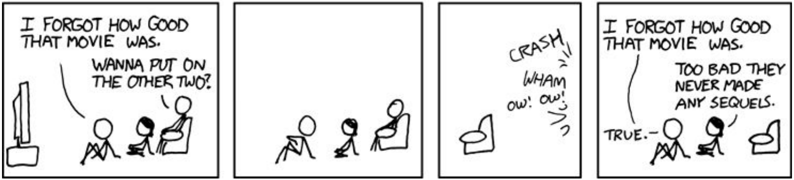
- Data partitioning with parallel matrix multiplication
- How to manage threads

Outline

Parallel Matrix Multiplication

Data partitioning based on

- Output matrix **C**
- Input matrix **A** and input matrix **B**



Source: http://www.xkcd.com

Output Partitioning (2 Threads)

```
// Thread 0
for (i=0; i<N/2; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
// Thread 1
for (i=N/2; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

Input Partitioning: Error in Last Week's Slides

```
// Thread 0
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N/2; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
// Thread 1
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=N/2; k<N; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

Error: Possible race condition when writing `c[i][j]`

Input Partitioning: Locking

```java
Object[][] lock; // Lock "matrix"
// Thread 0
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    synchronized(lock[i][j]) {
      for (k=0; k<N/2; k++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
// Thread 1
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    synchronized(lock[i][j]) {
      for (k=N/2; k<N; k++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

Overhead?

- A complete row is locked
- Actual lock contention will be moderate to low
- In practice the slow-down – with respect to output partitioning – is moderate (only a few percent)

Input Partitioning: Fine-Grain Locking

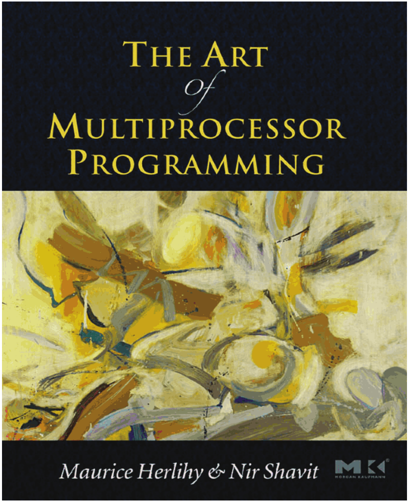
```
Object[][] lock; // Lock "matrix"
// Thread 0
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N/2; k++) {
      synchronized(lock[i][j]) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
// Thread 1
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=N/2; k<N; k++) {
      synchronized(lock[i][j]) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

Significant overhead:
About 3 times slower than coarse-grain locking

Outline

The Art of Multiprocessor Programming

Source of the following material about Thread Pools: “The Art of Multiprocessor Programming” by Maurice Herlihy and Nir Shavit



Thread Overhead

- Threads require resources
  - Memory for stacks
  - Setup, teardown
- Scheduler overhead
- Worse for short-lived threads

Thread Pools

- More sensible to keep a pool of long-lived threads
- Threads assigned short-lived tasks
  - Runs the task
  - Rejoins pool
  - Waits for next assignment

Thread Pool = Abstraction

- Insulate programmer from platform
  - Big machine, big pool
  - And vice-versa
- Portable code
  - Runs well on any platform
  - No need to mix algorithm/platform concerns

`ExecutorService` Interface

In `java.util.concurrent`:

- Task = `Runnable` object
  - If no result value expected
  - Calls `run()` method.
- Task = `Callable<T>` object
  - If result value of type `T` expected
  - Calls `T call()` method.

Future<T>

```
Callable<T> task = ...;
...
Future<T> future = executor.submit(task);
...
T value = future.get();
```

- Submitting a `Callable<T>` task returns a `Future<T>` object
- The Future's `get()` method blocks until the value is available

Future<?>

```
Runnable task = ...;
...
Future<?> future = executor.submit(task);
...
future.get();
```

- Submitting a `Runnable` task returns a `Future<?>` object
- The Future's `get()` method blocks until the computation is complete

Note

- Executor Service submissions are purely advisory in nature
- The executor
  - Is free to ignore any such advice
  - And could execute tasks sequentially ...

Fibonacci

$$F(n) := \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n > 1 \end{cases}$$

- Potential parallelism
- Dependencies

Disclaimer

- This Fibonacci implementation is very inefficient
  - So don't deploy it!
- But illustrates our point
  - How to deal with dependencies

Multithreaded Fibonacci

```java
class FibTask implements Callable<Integer> {
  static ExecutorService exec =
    Executors.newCachedThreadPool();
  int arg;
  public FibTask(int n) {
    arg = n;
  }
  public Integer call() {
    if (arg > 2) {
      Future<Integer> left =
        exec.submit(new FibTask(arg-1));
      Future<Integer> right =
        exec.submit(new FibTask(arg-2));
      return left.get() + right.get();
    } else {
      return 1;
    }
  }
}
```

Summary

Enjoy your vacation!