Programmazione di sistemi multicore

A.A. 2015-2016

LECTURE 5

IRENE FINOCCHI

http://wwwusers.di.uniroma1.it/~finocchi/

Hands on code

- IMPLEMENTATION
- MEASURING TIME
  - Real time
  - User time
  - System time
- OBSERVING CPU USAGE FOR DIFFERENT NUMBERS OF THREADS
- **MEMORY VS. CPU BOUNDED COMPUTATIONS**

Which memory is shared?

- Fork-join programs (thankfully) do not require much focus on sharing memory among threads
- But in languages like Java, there is memory being shared. In our example:
  - **`lo`**, **`hi`**, **`arr`** fields written by "main" thread, read by helper thread
  - **`ans`** field written by helper thread, read by "main" thread
- When using shared memory, you must avoid **data race conditions**
  - output depends on timing of other uncontrollable events
  - the order in which internal variables are changed determines the eventual state that the state machine will end up in

Join (not the most descriptive word)

- The `Thread` class defines various methods you could not implement on your own
  - For example: `start`, which calls `run` in a new thread
- The `join` method is valuable for coordinating this kind of computation
  - Caller blocks until/unless the receiver is done executing (meaning the call to `run` returns)
  - Else, we would have a race condition on `ts[i].ans`
  - While studying parallelism, we will stick with `join`
  - With concurrency, we will learn other ways to synchronize
- This style of parallel programming is called **"fork/join"**

A Java implementation detail

- Code has 1 compile error because `join` may throw `java.lang.InterruptedException`
  - Thrown when a thread is interrupted
  - Thread could be in either waiting, sleeping or running state and this exception can be thrown either before or during a thread's activity

- In basic parallel code, should be fine to catch-and-exit
  - We will throw the InterruptedException to the upper layer of the calling stack and let the upper layer handle it

Fork/join parallelism

- JAVA THREADS
- USING THREADS TO IMPLEMENT FORALL
- USING JOIN TO SYNCHRONIZE THREADS
- **HOW MANY THREADS?**

Code portability

Several reasons why this is a poor parallel implementation

1. Want code to be reusable and efficient **across platfo**[illegible]
   - "Forward-portable" as core count grows
   - So, at the *very* least, parameterize by the n[illegible] of threads

```
int sum(int[] arr, int numTs){
  int ans = 0;
  SumThread[] ts = new Su[illegible]d[numTs];
  for(int i=0; i < numT[illegible]){
   ts[i] = new SumTh[illegible]rr,(i*arr.length)/numTs,
                              ((i+1)*arr.length)/numTs);
   ts[i].start([illegible]
  }
  for(int [illegible] < numTs; i++) {
    ts[[illegible]n();
    ans [illegible] ts[i].ans;
  }
  return ans;
}
```

Do not use constants where a variable is appropriate

Threads vs. processors/cores

2. Want to use (only) processors **available to you *now***
   - Not used by other programs or threads in your program
     - Maybe caller is also using parallelism
     - Available cores can change even while your threads run
   - With 3 processors available, using 3 threads would take time **`X`**, but creating 4 threads would take time **`1.5X`**!
     - Example: 12 units of work, 3 processors
       - Work divided into 3 parts will take 4 units of time
       - Work divided into 4 parts will take 3*2 units of time

```
// numThreads == numProcessors is bad
// if some are needed for other things
int sum(int[] arr, int numTs){
  …
}
```

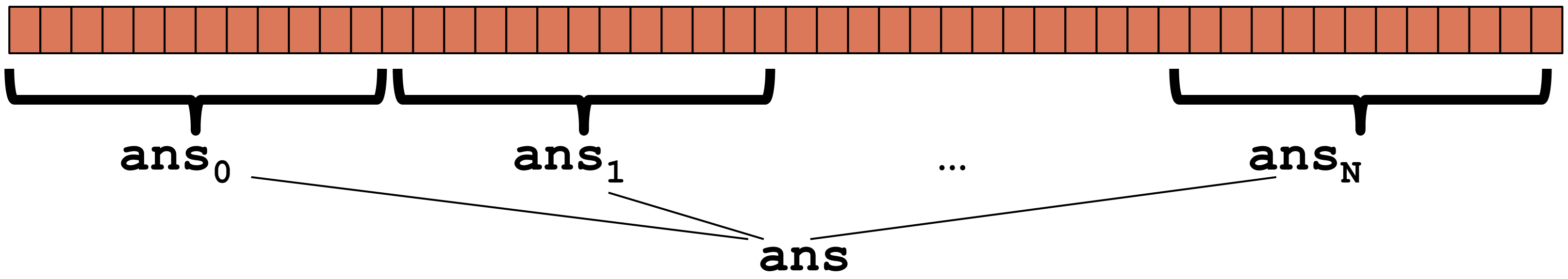
Load balancing

3. Though unlikely for **`sum`**, in general subproblems may take significantly different amounts of time
   - Typical scenario: Apply a method **`f`** to every array element, but **`f`** much slower for some data items
     - Example: given a large int[], how many elements are *prime numbers*? Checking that a number is prime could take longer than discovering it is not:
       - need to *check all possible divisors* in the former case
       - stop as soon as you find a divisor in the latter case
   - If we create 4 threads and all the slow data is processed by 1 of them, we won't get nearly a 4x speedup
     - Example of a **load imbalance**: different helper threads get different amounts of work

A counterintuitive solution!

Use lots of threads,
far more than the number of cores

- We'll see that this will require changing our algorithm
- And, for constant-factor reasons, abandoning Java's threads



Previous issues are solved…

- Forward-portable: code independent of #processors
  - Lots of helpers, each doing a small piece of work
- Processors available: just a "big pile" of threads waiting to run
  - If #processors available changes, that affects only how fast the pile is processed, but we are always doing useful work with available resources
  - Example: 120 units of work, 3 or 4 processors. Work divided into X parts (threads) will take X_p units of time on p processors:
    - X=3 part lenght=40 $X_3=40*1$ **$X_4=40*1$**
    - X=4 part lenght=30 **$X_3=30*2$** $X_4=30*1$
    - X=60 part lenght=2 $X_3=2*20$ $X_4=2*15$
    - X=120 part lenght=1 $X_3=1*40$ $X_4=1*30$
- Load balancing: Small pieces of work yields shorter threads
  - Slow threads are no problem if scheduled early enough

… but new issues arise

- Suppose we create 1 thread to process every 1000 elements
- Then combining results will have `arr.length / 1000` additions
  - Linear in size of array: Θ(arr.length) time, with constant factor 1/1000
  - Previously Θ(1) time
- In the extreme case, if we create 1 thread per element, the loop to combine partial results has `arr.length` iterations
  - Just like the original sequential algorithm!

```
int sum(int[] arr){
  …
  int numThreads = arr.length / 1000;
  SumThread[] ts = new SumThread[numThreads];
  …
  for(int i=0; i < numThreads; i++) {
    ts[i].join();
    ans += ts[i].ans;
  }
}
```

… but new issues arise

- Suppose we create 1 thread to process every 1000 elements
- Then combining results will have `arr.length / 100`… additions
  - Linear in size of array (Θ(arr.length) time, with … factor 1/1000)
  - Previously Θ(1) time
- In the extreme case, if we create 1 threa… …oop to combine partial results has `arr.l`…
  - Just like the original seque…

```
int sum(int[]
  …
  int nu…      …ength / 1000;
  SumT…          SumThread[numThreads];
  …
  for(i…    < numThreads; i++) {
    ts[i]…n();
    ans +… ts[i].ans;
  }
}
```

- Need to use a **clever algorithmic approach**
- Java's threads are not designed for small tasks: too many threads yield large **overhead**

Lecture recap

- Hands on code (again)
  - Memory vs CPU-bounded computations: practical implications on speedup
  - Using a profiler (-prof) to determine percentage of sequential and parallel code

- Choosing the appropriate number of threads
  - Forward portable code
  - Threads vs (available) processors/cores
  - Load balancing
  - Theoretical and practical issues with too may threads

Sources

- A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency, Dan Grossman

  http://www.cs.washington.edu/homes/djg/teachingMaterials