

CSE 332: Data Structures and Parallelism

Spring 2022

Richard Anderson

Lecture 15: Introduction to Parallelism

Course Schedule

- Lectures 1- 14: Traditional Data Structures
- Lectures 15-19: Parallelism
- Lectures 20-22: Concurrency
- Lectures 23-27: Graph Algorithms
- Lectures 28-29: Theory of NP-Completeness

Announcements

- Read parallel computing notes by Dan Grossman 2.1-3.4
- Midterm – still underway

Sequential Summation

- Sum up N numbers in an array
– Complexity?



Parallel Sum

- Sum up N numbers in an array
– with two processors



Parallel Sum

- Sum up N numbers in an array
– with ten processors



Parallel Sum

- Sum up N numbers in an array
 - with N processors?



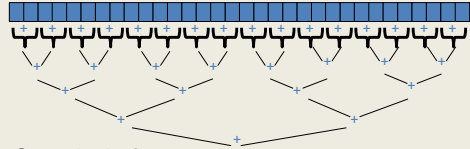
5/2/2022

CSE 332

7

Parallel Sum

- Sum up N numbers in an array



- Complexity?
- How many processors?
- Faster with infinite processors?

5/2/2022

CSE 332

8

Parallel Algorithms

- So far, we have assumed:
 - One thing happens at a time*
- What if we want to implement algorithms with multiple “processors”
 - How do we model parallel computing
 - How do we program parallel computers

5/2/2022

CSE 332

9

Parallel Computation

- There is nothing new about parallel computation
- Hardware design and architecture have always been about parallelism
 - Parallelism has been central to computer performance
- Parallel algorithms have been an area of study since the late 1970s
- Hardware trends
 - Multiple cores in processors
 - Can no longer make components smaller to make them faster – need to make more of them

5/2/2022

CSE 332

10

Who Implements Parallelism

- User
- Application
- Operating System
- Programming Language, Compiler
- Algorithm
- Processor Hardware

5/2/2022

CSE 332

11

Parallelism vs. Concurrency

Parallelism:
Use extra resources to solve a problem faster



Concurrency:
Manage access to shared resources



5/2/2022

CSE 332

12

Shared Memory with Threads

Old story: A running program has

- One *program counter* (current statement executing)
- One *call stack* (with each *stack frame* holding local variables)
- *Objects in the heap* created by memory allocation (i.e., **new**)
 - (nothing to do with data structure called a heap)
- *Static fields*

New story:

- A set of *threads*, each with its own program counter & call stack
 - No access to another thread's local variables
- Threads can share static fields / objects
 - To *communicate*, write values to some shared location that another thread reads from

5/2/2022

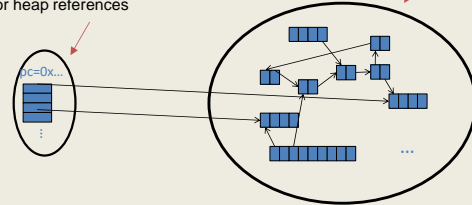
CSE 332

13

Old Story: one call stack, one pc

- Call stack with local variables
- pc determines current statement
- local variables are numbers/null or heap references

Heap for all objects and static fields



5/2/2022

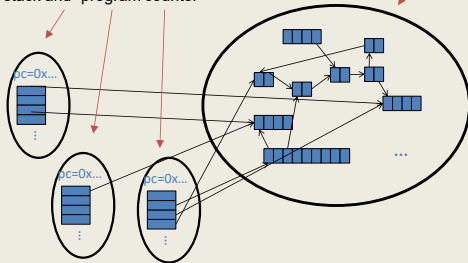
CSE 332

14 14

New Story: Shared Memory with Threads

Threads, each with own *unshared* call stack and "program counter"

Heap for all objects and static fields, *shared* by all threads



5/2/2022

CSE 332

15

Other models

We will focus on shared memory, but you should know several other models exist and have their own advantages (**see notes**)

- **Message-passing:** Each thread has its own collection of objects. Communication is via explicitly sending/receiving messages
- **Dataflow:** Programmers write programs in terms of a DAG. A node executes after all of its predecessors in the graph
- **Data parallelism:** Have primitives for things like "apply function to every element of an array in parallel"

5/2/2022

CSE 332

16

Our Needs

To write a shared-memory parallel program, need new primitives from a programming language or library

- Ways to create and *run multiple things at once*
 - Let's call these things **threads**
- Ways for threads to *share memory*
 - Often just have threads with references to the same objects
- Ways for threads to *coordinate (a.k.a. synchronize)*
 - For now, a way for one thread to wait for another to finish
 - Other primitives when we study concurrency

5/2/2022

CSE 332

17

Threads vs. Processors

What happens if you start 5 threads on a machine with only 4 processors?

5/2/2022

CSE 332

18

Threads vs. Processors

For sum operation:

- with 3 processors available, using 4 threads would take 50% more time than 3 threads

5/2/2022

CSE 332

19

Fork-Join Parallelism

1. Define thread

- Java: define subclass of `java.lang.Thread`, override `run`

2. Fork: instantiate a thread and start executing

- Java: create thread object, call `start()`

3. Join: wait for thread to terminate

- Java: call `join()` method, which returns when thread finishes

Above uses basic thread library build into Java

Later we'll introduce a better `ForkJoin Java library` designed for parallel programming

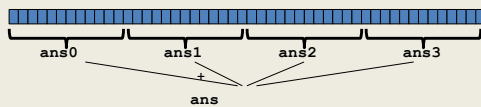
5/2/2022

CSE 332

20

Sum with Threads

For starters: have 4 threads simultaneously sum ¼ of the array



- Create 4 *thread objects*, each given ¼ of the array
- Call `start()` on each thread object to run it in parallel
- Wait for threads to finish using `join()`
- Add together their 4 answers for the final result

5/2/2022

CSE 332

21

Part 1: define thread class

```
class SumThread extends java.lang.Thread {  
    int lo; // fields, passed to constructor  
    int hi; // so threads know what to do.  
    int[] arr;  
  
    int ans = 0; // result  
  
    SumThread(int[] a, int l, int h) {  
        lo=l; hi=h; arr=a;  
    }  
  
    public void run() { //override must have this type  
        for(int i=lo; i < hi; i++)  
            ans += arr[i];  
    }  
}
```

Because we must override a no-arguments/no-result run, we use fields to communicate across threads

5/2/2022

CSE 332

22

Part 2: sum routine

```
int sum(int[] arr){ // can be a static method  
    int len = arr.length;  
    int ans = 0;  
    SumThread[] ts = new SumThread[4];  
    for(int i=0; i < 4; i++){ // do parallel computations  
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);  
        ts[i].start();  
    }  
    for(int i=0; i < 4; i++) { // combine results  
        ts[i].join(); // wait for helper to finish!  
        ans += ts[i].ans;  
    }  
    return ans;  
}
```

5/2/2022

CSE 332

23

Parameterizing by number of threads

```
int sum(int[] arr, int numTs){  
    int ans = 0;  
    SumThread[] ts = new SumThread[numTs];  
    for(int i=0; i < numTs; i++){  
        ts[i] = new SumThread(arr, (i*arr.length)/numTs,  
                                ((i+1)*arr.length)/numTs);  
        ts[i].start();  
    }  
    for(int i=0; i < numTs; i++) {  
        ts[i].join();  
        ans += ts[i].ans;  
    }  
    return ans;  
}
```

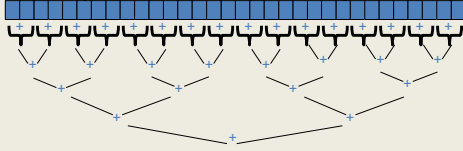
5/2/2022

CSE 332

24

Recall: Parallel Sum

- Sum up N numbers in an array



- Let's implement this with threads...

5/2/2022

CSE 332

25

Code looks something like this (using Java Threads)

```
class SumThread extends java.lang.Thread {
    int lo; int hi; int[] arr; // fields to know what to do
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { // override
        if (hi - lo < SEQUENTIAL_CUTOFF)
            for (int i=lo; i < hi; i++)
                ans += arr[i];
        else {
            SumThread left = new SumThread(arr, lo, (hi+lo)/2);
            SumThread right = new SumThread(arr, (hi+lo)/2, hi);
            left.start();
            right.start();
            left.join(); // don't move this up a line - why?
            right.join();
            ans = left.ans + right.ans;
        }
    }
}

int sum(int[] arr) { // just make one thread!
    SumThread t = new SumThread(arr, 0, arr.length);
    t.run();
    return t.ans;
}
```

5/2/2022

CSE 332

26

Recursive problem decomposition

```
Thread: sum range [0,10]
  Thread: sum range [0,5]
    Thread: sum range [0,2]
      Thread: sum range [0,1] (return arr[0])
      Thread: sum range [1,2] (return arr[1])
      add results from two helper threads
    Thread: sum range [2,5]
      Thread: sum range [2,3] (return arr[2])
      Thread: sum range [3,5]
        Thread: sum range [3,4] (return arr[3])
        Thread: sum range [4,5] (return arr[4])
        add results from two helper threads
      add results from two helper threads
    add results from two helper threads
  Thread: sum range [5,10]
    Thread: sum range [5,7]
      Thread: sum range [5,6] (return arr[5])
      Thread: sum range [6,7] (return arr[6])
      add results from two helper threads
    Thread: sum range [7,10]
      Thread: sum range [7,8] (return arr[7])
      Thread: sum range [8,10]
        Thread: sum range [8,9] (return arr[8])
        Thread: sum range [9,10] (return arr[9])
        add results from two helper threads
      add results from two helper threads
    add results from two helper threads
```

5/2/2022

CSE 332

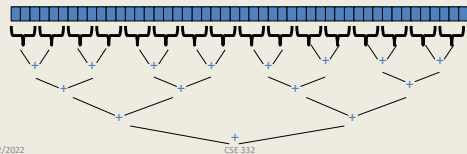
27

Divide-and-conquer

Same approach useful for many problems beyond sum

- If you have enough processors, total time $O(\log n)$
- Next lecture: study reality of $P \ll n$ processors

- Will write all our parallel algorithms in this style
 - But using a special fork-join library engineered for this style
 - Takes care of scheduling the computation well
 - Often relies on operations being associative (like +)



5/2/2022

CSE 332

28

Thread Overhead

Creating and managing threads incurs cost

Two optimizations:

- Use a *sequential cutoff*, typically around 500-1000
 - Eliminates lots of tiny threads
- Do not create two recursive threads; create one thread and do the other piece of work "yourself"
 - Cuts the number of threads created by another 2x

5/2/2022

CSE 332

29

Half the threads!

order of last 4 lines

Is critical - why?

```
// wasteful: don't
SumThread left = ...
SumThread right = ...

left.start();
right.start();
```

```
// better: do!!
SumThread left = ...
SumThread right = ...
```

```
left.start();
right.run();
```

Note: `run` is a normal function call; execution won't continue until we are done with `run`

```
left.join();
right.join();
ans=left.ans+right.ans;
```

```
left.join();
// no right.join needed
ans=left.ans+right.ans;
```

5/2/2022

CSE 332

30

Better Java Thread Library

- Even with all this care, Java's threads are too "heavyweight"
 - Constant factors, especially space overhead
 - Creating 20,000 Java threads just a bad idea ☹
- The **ForkJoin Framework** is designed to meet the needs of divide-and-conquer fork-join parallelism
 - In the Java 8 standard libraries
 - Section will focus on pragmatics/logistics
 - Similar libraries available for other languages
 - C/C++: Cilk (inventors), Intel's Thread Building Blocks
 - C#: Task Parallel Library
 - ...

5/2/2022

CSE 332

31

Different terms, same basic idea

To use the ForkJoin Framework:

- A little standard set-up code (e.g., create a **ForkJoinPool**)

Don't subclass Thread	Do subclass RecursiveTask<V>
Don't override run	Do override compute
Do not use an ans field	Do return a V from compute
Don't call start	Do call fork
Don't <i>just</i> call join	Do call join (which returns answer)
Don't call run to hand-optimize	Do call compute to hand-optimize
Don't have a topmost call to run	Do create a pool and call invoke

See the web page for (linked in to project 3 description):

"A Beginner's Introduction to the ForkJoin Framework"

5/2/2022

CSE 332

32

Fork Join Framework Version: (missing imports)

```
class SumArray extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; // fields to know what to do
    SumArray(int[] a, int l, int h) { ... }
    protected Integer compute() { // return answer
        if (hi - lo < SEQUENTIAL_CUTOFF) {
            int ans = 0; // local var, not a field
            for (int i = lo; i < hi; i++)
                ans += arr[i];
            return ans;
        } else {
            SumArray left = new SumArray(arr, lo, (hi+lo)/2);
            SumArray right = new SumArray(arr, (hi+lo)/2, hi);
            left.fork(); // fork a thread and calls compute
            int rightAns = right.compute(); // call compute directly
            int leftAns = left.join(); // get result from left
            return leftAns + rightAns;
        }
    }
}

static final ForkJoinPool fjPool = new ForkJoinPool();
int sum(int[] arr) {
    return fjPool.invoke(new SumArray(arr, 0, arr.length));
    // invoke returns the value compute returns
}
```

5/2/2022

CSE 332

33