parallel concurrent

Amdahl's Law thread of control

processors versus processes

non-deterministic fork-join parallelism.

Parallel and concurrent programming

4. Parallel programming in Java

protection data race synchronization

divide-and-conquer algorithms

THREAD

SAFETY correctness

Michelle Kuttel

MUTUAL EXCLUSION

locks

readers-writers problem

liveness

DEADLOCK starvation

HIGH PERFORMANCE COMPUTING

producer-consumer problem

EXECUTORS thread pools timing

Dining philosphoers problem

Basic Parallel Problem: summing the elements of a large array

(This problem is to illustrate the concept of parallelization, it's *not* an ideal problem to parallelize.)

An **O(n)** sequential solution to this problem is trivial:

```
int sum(int[] arr) {
    int ans = 0;
    for(int i=0; i < arr.length; i++)
        ans += arr[i];
    return ans;
}
```

Parallel programming is only really worth the effort for programs that take too long to run serially…

Time the sequential/serial solution **as a *benchmark***

Just fill a big array with 1's and see how long it takes to add. Do the sum a few times and time each (to check for cache effects). e.g.

```
Adding 100000 integers serially took 2.0 milliseconds
Adding 100000 integers serially took 1.0 milliseconds
Adding 100000 integers serially took 1.0 milliseconds
```

Or…

```
Adding 100000000 integers serially took 37.0 milliseconds
Adding 100000000 integers serially took 34.0 milliseconds
Adding 100000000 integers serially took 35.0 milliseconds
```

Or…

```
Adding 100000000 doubles serially took 110.0 milliseconds
Adding 100000000 doubles serially took 114.0 milliseconds
Adding 100000000 doubles serially took 102.0 milliseconds
```

Parallel programming is only really worth the effort for programs that take too long to run serially…

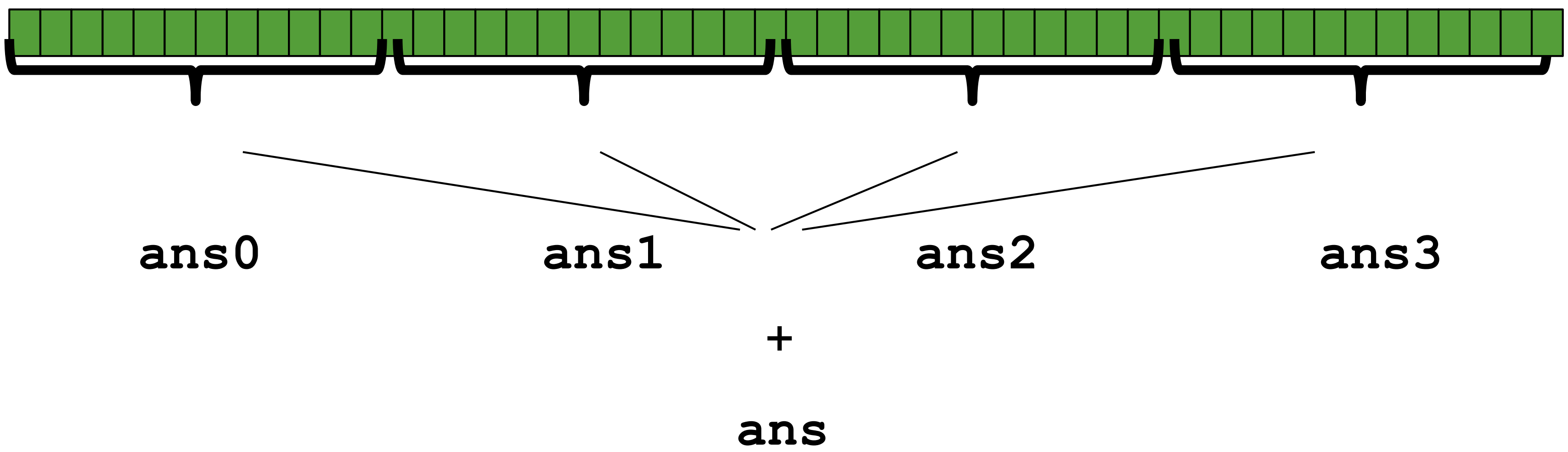
```
System.currentTimeMillis();
```

Now for a parallel version…

Parallelism idea 1 with normal threads: Okay Idea, poor Style

Suppose we have 4 processors/cores/cpus.

- Idea: Have 4 threads simultaneously sum 1/4 of the array each
  - **Warning: poor first approach**



Parallelism idea 1: Okay Idea, Inferior Style

`In Java`

- Create 4 *thread objects*, give each a portion of the work

- Call **`start()`** on each thread object to actually *run* it in parallel

- *Wait* for threads to finish using **`join()`**

- Add together their 4 answers for the *final result*

First attempt, part 1

```
class SumThread extends java.lang.Thread {

  int lo; // arguments
  int hi;
  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` method, we use fields/variables to communicate across threads

First attempt, continued (wrong)

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

Want code to be reusable and efficient across platforms

-> “scalable” as core count grows

Therefore, **parameterize** by the **number of threads** -

- For **P** processors, divide the array into **P** equal segments
- algorithm runs in time **O(n/P + P)** where n/P is the parallel part and P is for combining the stored results.

First attempt, continued (wrong)

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

WHAT IS WRONG?

slide adapted from: Sophomoric Parallelism and Concurrency, Lecture 1

First attempt, continued (wrong)

look at output....

```
public static void main(String[] args) {
            int max =100000;
            int noThreads =4;
            int [] arr = new int[max];
            for (int i=0;i<max;i++) { arr[i]=10000;}
            int sumArr = sum(arr,noThreads);
            System.out.println("Sum is:");
            System.out.println(sumArr);
      }
```

Second attempt (still wrong)

```
static int sum(int[] arr, int numTs) throws
      InterruptedException {
   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();  //start, not run
   }
   for(int i=0; i < numTs; i++) {
      ans += ts[i].ans;
   }
    return ans;
}
```

Why still wrong?

Second attempt (still wrong)

```
static int sum(int[] arr, int numTs) throws
      InterruptedException {
   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(); //start, not run
   }
   for(int i=0; i < numTs; i++) {
      ans += ts[i].ans;
   }
    return ans;
}
```

look at output.... race condition.

Second attempt (still wrong)

```java
static int sum(int[] arr, int numTs) throws
        InterruptedException {
    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(); //start, not run
    }
    for(int i=0; i < numTs; i++) {
        ans += ts[i].ans;
    }
     return ans;
}
```

`lo`, `hi`, `arr` fields written by "main" thread, read by helper thread

`ans` field written by helper thread, read by "main" thread

race condition on `ts[i].ans`

Third attempt (correct in spirit)

```
static int sum(int[] arr, int numTs) throws
InterruptedException {
  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();  //start, not run
  }
  for(int i=0; i < numTs; i++) {
            ts[i].join(); // wait for helper to finish!
      ans += ts[i].ans;
  }
  return ans;
}
```

Join (again)

This style of parallel programming is called "**fork/join parallelism**"

Third attempt (correct in spirit)

```
int sum(int[] arr) throws InterruptedException{// 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;
}
```

```
public static void main(String[] args) {
 // [….]
  try {
        int sumArr = sum(arr,noThreads);
        System.out.println("Sum is:");
        System.out.println(sumArr);
  } catch (InterruptedException e) {
        e.printStackTrace();
}
```

`join` may throw **`java.lang.InterruptedException`** should be fine to catch-and-exit

For concurrent programming, it may be bad style to ignore this exception, but for basic parallel programming like we are doing, this exception is a nuisance and will not occur.

Need to experiment with number of threads

- Get from system?

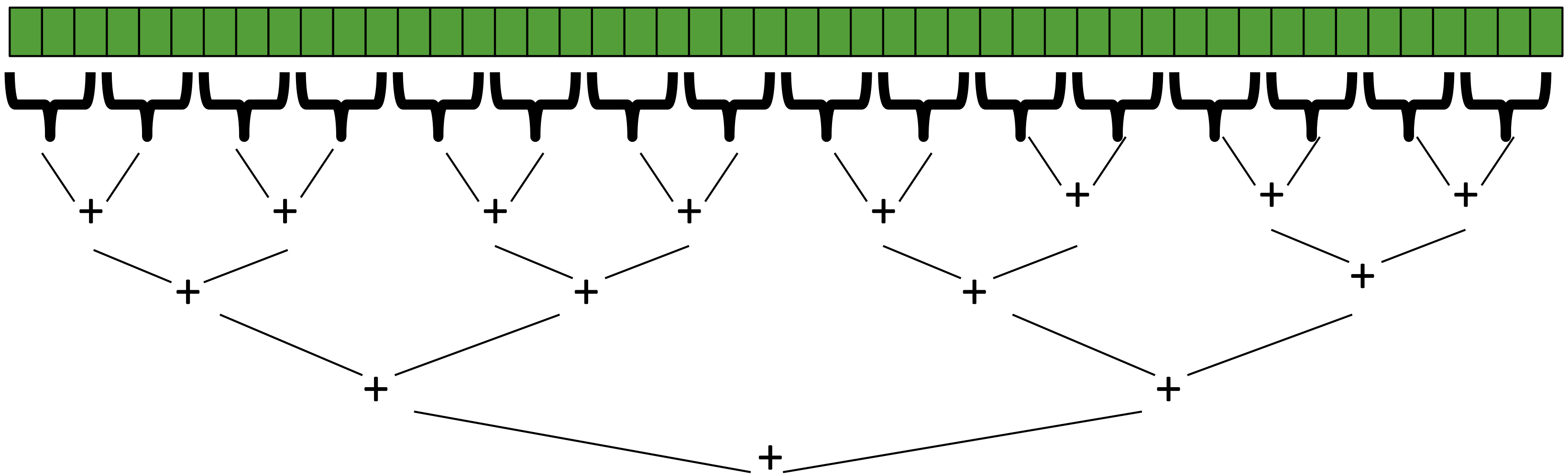
```
int noThreads = Runtime.getRuntime().availableProcessors();
```

- Will this be optimal?
- It depends….

Timing to benchmark on my laptop (10 logical cores)

- 8-core CPU with 6 performance cores and 2 efficiency cores

Alternative approach: use Fork/Join



This is straightforward to implement using divide-and-conquer

- Parallelism for the recursive calls

The **result-combining** is done in **parallel** as well

- more efficient
- *If* you have enough processors, total time is height of the tree:

$O($`log` $n)$ (optimal, exponentially faster than sequential $O(n)$)

What you need to know about the library

ForkJoinTasks (either RecursiveAction or RecursiveTask) are given to a **ForkJoinPool** (a pool of threads).

You can create the pool

- **static final** ForkJoinPool ***fjPool*** = **new** ForkJoinPool();
  - **ForkJoinPool()** creates a ForkJoinPool with **parallelism** equal to Runtime.availableProcessors().
  - You can specify the "parallelism"

or use the default one

- **static final** ForkJoinPool fjPool = ForkJoinPool.***commonPool*()**;
  - commonPool() is static, always available and appropriate for most applications. Has **parallelism** equal to Runtime.availableProcessors() -1.

How many threads? – by default equal to the "parallelism", but you can change this… up to a maximum (32767)

Example: final F/J version (missing imports)

```
public class SumArray extends RecursiveTask<Integer>  {
  int lo; // arguments
  int hi;
  int[] arr;
  static final int SEQUENTIAL_CUTOFF=5000;
  int ans = 0; // result

  SumArray(int[] a, int l, int h) {
   lo=l; hi=h; arr=a;
  }

  protected Integer compute(){// return answer - instead of
run
  if((hi-lo) < SEQUENTIAL_CUTOFF) {
  int ans = 0;
     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(); //this
      int rightAns = right.compute(); //order
      int leftAns  = left.join();   //is very
      return leftAns + rightAns;      //important.
   }}}
```

```
public class SumAll {
static final ForkJoinPool fjPool = new
ForkJoinPool();
static int sum(int[] arr){
  return fjPool.invoke(new
SumArray(arr,0,arr.length));
}

public static void main(String[] args) {
int max =100000000;
int [] arr = new int[max];
for (int i=0;i<max;i++) {
arr[i]=1;
}
int sumArr = sum(arr);
System.out.println("Sum is:" + sumArr);
}
```

Half the threads

Don't create two recursive threads; create one and do the other "yourself"

- Cuts the number of threads created by another 2x

This won't be tested, but I should understand it

```
// wasteful: don't
SumArray left = new
          SumArray(arr,lo,(hi+lo)/2);
SumArray right= new
          SumArray(arr,(hi+lo)/2,hi);
left.fork(); //this
right.fork();
int leftAns  = left.join();
int rightAns  = right.join();
return leftAns + rightAns;
```

```
// better: do
SumArray left = new
          SumArray(arr,lo,(hi+lo)/2);
SumArray right= new
          SumArray(arr,(hi+lo)/2,hi);
left.fork(); //this
int rightAns = right.compute();
int leftAns  = left.join();
return leftAns + rightAns;
```

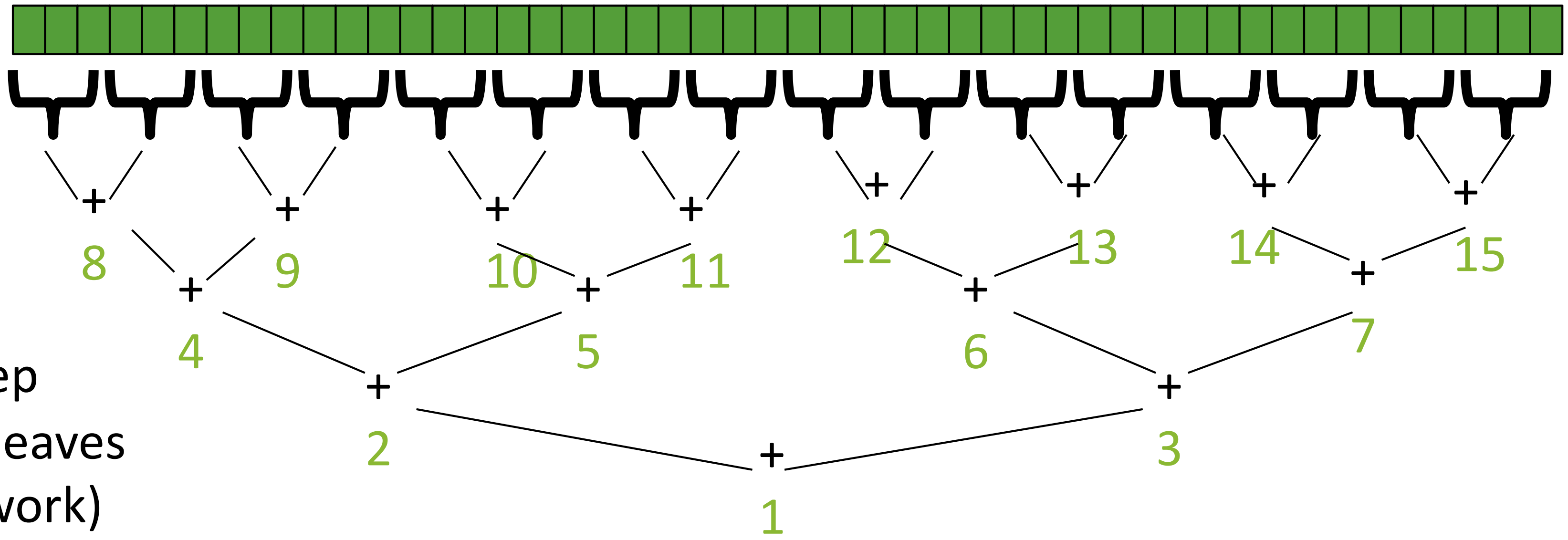
Sequential cut-off for tasks

In theory, you can divide down to single elements, do all your result-combining in parallel and get optimal speedup

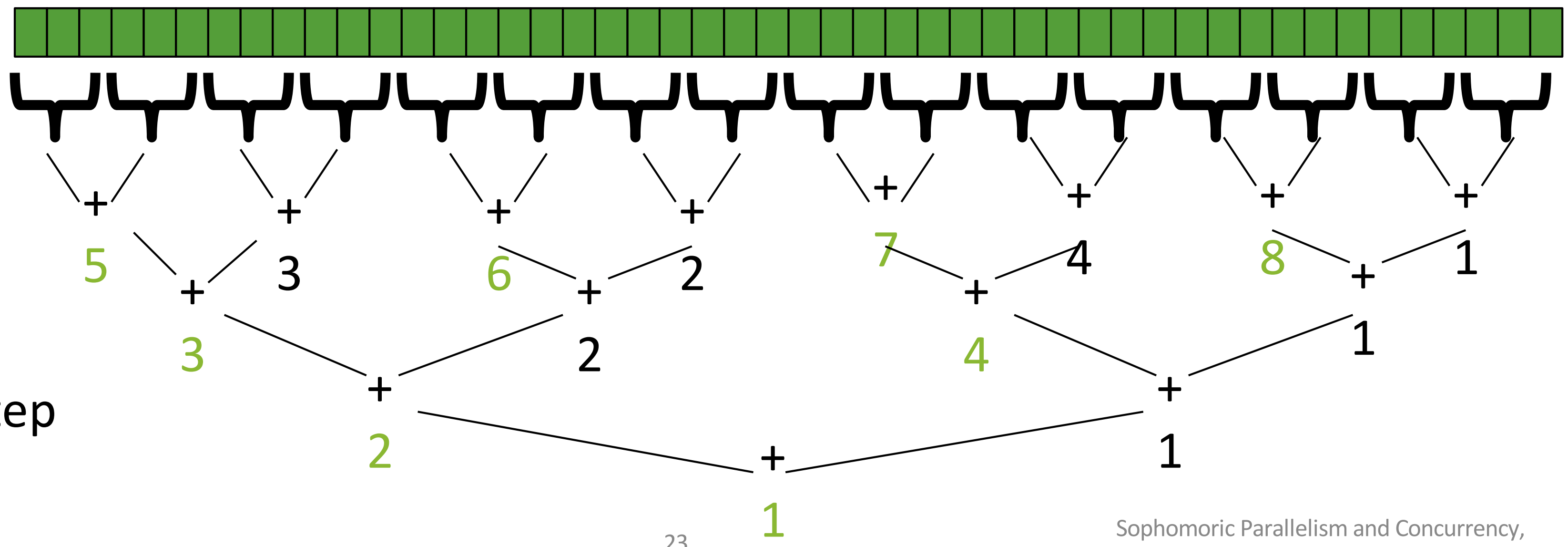
- Total time $O(n/P + \log n)$

- In practice, there is a point where the *fork* costs more than the calculation so:
  - Use a *sequential cutoff*, (value **depends on the algorithm**)
    - *Exactly* like quicksort switching to insertion sort for small subproblems, but more important here

Fewer threads pictorially



2 new threads at each step (and only leaves do much work)



1 new thread at each step

Compare times for the two versions

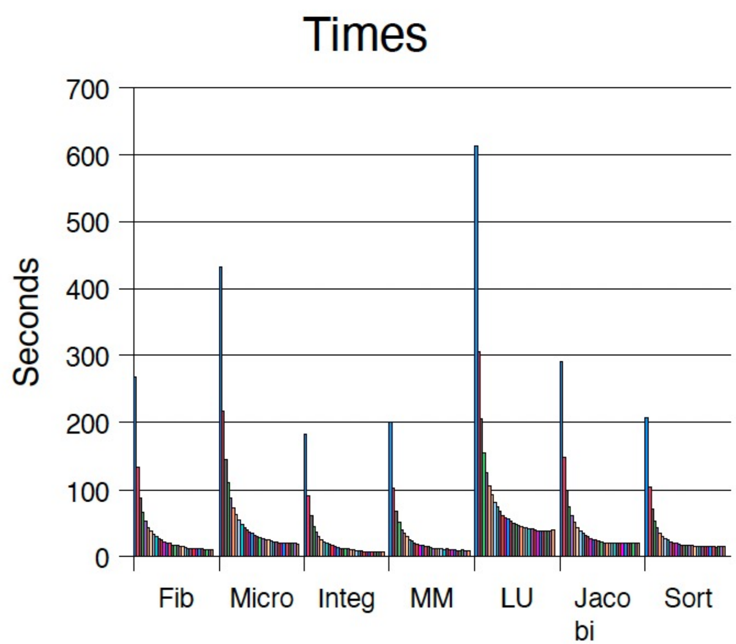
- Must use big enough data sets… why?

Note for Assignment 1. I must use big data set

Why such different run times?

The F/J framework needs to “warm up”

- May see slow results before the Java virtual machine re-optimizes the library internals
- Put your computations in a loop to see the “long-term benefit”
- need to do multiple timings
  - Avoid warm−up by running an initial problem set before timing



From:
A Java Fork/Join Framework
Doug Lea
State University of New York

When Fork/Join is really useful

- When you are doing the parallel computation many times
- When threads have a **lot to do**
- When threads have different amounts of work to do – **load imbalance**
  - Though unlikely for `sum`, in general sub problems may take significantly different amounts of time
  - Example: Apply method `f` to every array element, but maybe `f` is much slower for some data items
    - Example: Is a large integer prime?

`FJframework` provides "nearly ideal speedups for nearly any fork/join program on commonly available 2−way, 4−way, and 8−way SMP machines."

From:
A Java Fork/Join Framework
Doug Lea
State University of New York

Beware – things that are time consuming

Avoid:

- Creating new arrays every iteration – rather re-use arrays
- For the same reason, try not to re-create variables in a loop