parallel concurrent

Amdahl's Law thread of control

processors versus processes

non-deterministic fork-join parallelism.

Parallel and concurrent programming

5. Parallel algorithms

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

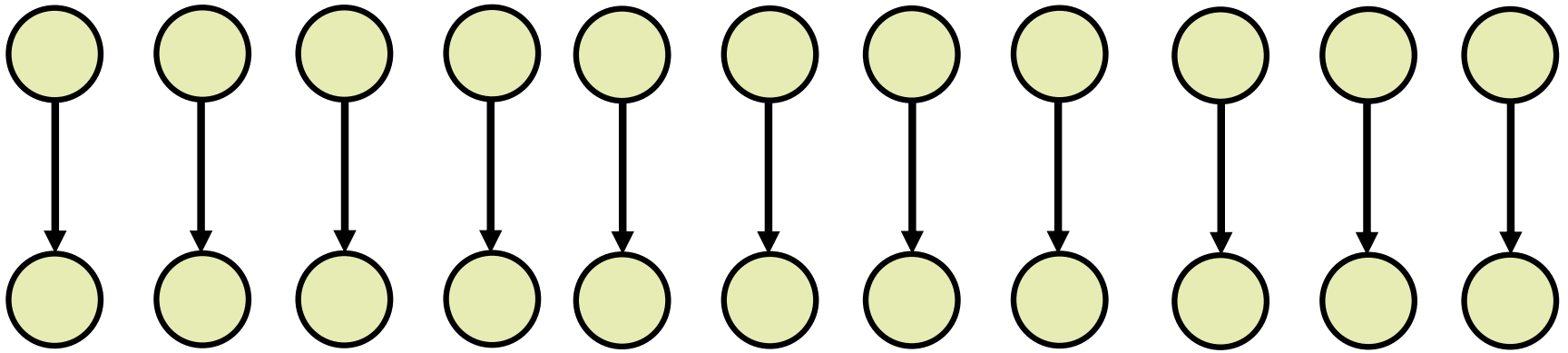
EXECUTORS thread pools timing

Dining philosphoers problem

First: embarrassingly parallel algorithms

Ideal computation - a computation that can be divided into a number of completely separate tasks, each of which can be executed by a single processor.

- **little to no communication or synchronization** between tasks
- no special algorithms or techniques required to get a workable solution



Divide-and-conquer parallel algorithms

Divide problems into sub problems that are of the same form as the larger problem

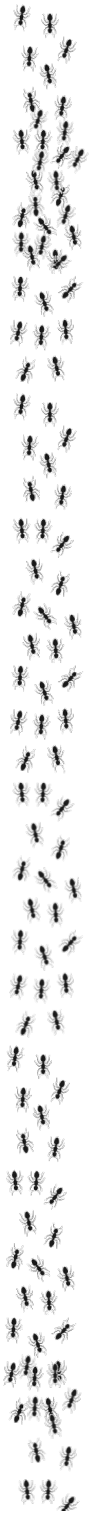
1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Recursive subdivision continues until the grain size of the problem is small enough to be solved sequentially.

It is **straight-forward** to convert an embarrassingly parallel algorithm into a divide-and-conquer parallel algorithm.

Examples of divide and conquer algorithms

- Finding maximum or minimum element
- Is there an element satisfying some property
  - e.g., is there a 17?
- Left-most element satisfying some property
  - e.g., first 17
  - *What should the recursive tasks return?*
  - *How should we merge the results?*
- Corners of a rectangle containing all points (a "bounding box")
- Counts, e.g. number of strings that start with a vowel
  - This is **just summing** with a different base case
  - Many problems are!

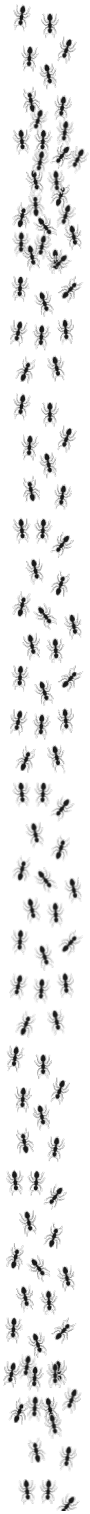


Basic Divide-and-Conquer algorithms: Reductions

- **Reduction** operations produce a single answer from collection via an **associative operator**
  - Examples: max, count, leftmost, rightmost, **sum**, …
  - Non-example: median

- Note: (Recursive) results don't have to be single numbers or strings. They can be arrays or objects with multiple fields.
  - Example: **Histogram** of test results is a variant of sum

- But some things are inherently sequential
  - How we process **`arr[i]`** may depend on the result of processing **`arr[i-1]`**



Basic divide and conquer algorithms: Maps

A **map** operates on each element of a collection independently to create a new collection of the same size

- No combining results
- For arrays, this is so trivial some hardware has direct support

Canonical example: Vector addition (*pseudo-code*)

```
int[] vector_add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);

  result = new int[arr1.length];

  FORALL(i=0; i < arr1.length; i++) {
    result[i] = arr1[i] + arr2[i];
  }
  return result;
}
```

I have to understand this code, for test & exam

Maps in ForkJoin Framework

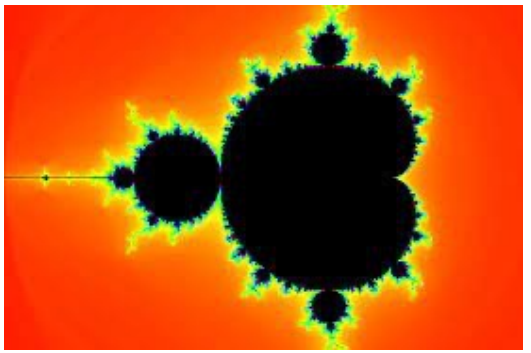
```
class VecAdd extends RecursiveAction {
  int lo; int hi; int[] res; int[] arr1; int[] arr2;
  VecAdd(int l,int h,int[] r,int[] a1,int[] a2){ … }
  protected void compute(){
    if(hi – lo < SEQUENTIAL_CUTOFF) {
              for(int i=lo; i < hi; i++)
        res[i] = arr1[i] + arr2[i];
    } else {
      int mid = (hi+lo)/2;
      VecAdd left = new VecAdd(lo,mid,res,arr1,arr2);
      VecAdd right= new VecAdd(mid,hi,res,arr1,arr2);
      left.fork();
      right.compute();
      left.join();
    }
  }
}
static final ForkJoinPool fjPool = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  fjPool.invoke(new VecAdd(0,arr.length,ans,arr1,arr2);
  return ans;
}
```

from: Sophomoric Parallelism and Concurrency, Lecture 2

Example – parallelising Image Processing algorithms

Low-level image processing uses the **individual pixel** values to modify the image in some way.

- Image processing operations can be divided into:
  - **point processing** – output produced based on value of single pixel
    - shift, rotate, clip, scale
    - colouring pixels according to well known **Mandelbrot set**



  - **local operations** – produce output based on a group of neighbouring pixels – e.g smoothing

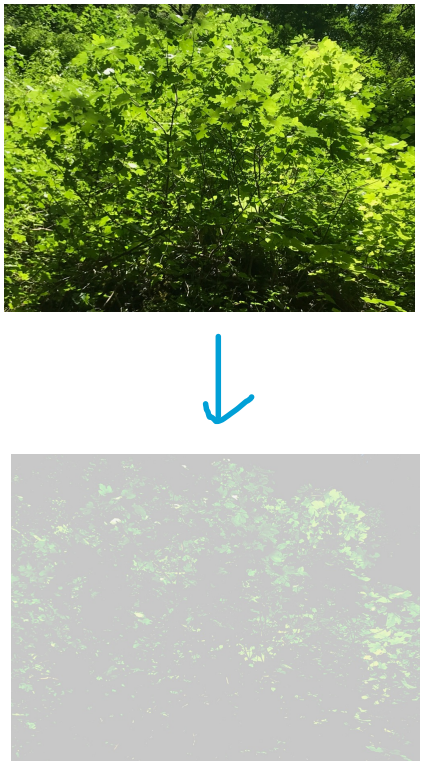
smoothing

Map – image threshholding

```
public class ParallelTest1 extends RecursiveAction {
    private BufferedImage img;
    private int x_start, x_stop;
    private int y_start, y_stop;
    private BufferedImage dstImg;
    protected static int sThreshold = 10000;

    public ParallelTest1(BufferedImage img,  int x_strt, int x_stp, int y_strt,  int y_stp,
BufferedImage dst) {
        this.img = img;
        x_start = x_strt; x_stop= x_stp;
        y_start = y_strt; y_stop= y_stp;
        this.dstImg = dst;
    }

  protected void work() {
        int R, G, B, clr;
        for (int y = y_start; y < y_stop; y++) {
            for (int x = x_start; x < x_stop; x++) {
                clr   = img.getRGB(x, y);  //fix this
                R = (clr>>16) & 0xff;
                G= (clr>> 8) & 0xff;
                B= (clr)& 0xff;
                R=Math.max(R,200);
                G=Math.max(G,200);
                B=Math.max(B,200);
                clr = (0xff000000) | (R << 16) | (G<< 8) | B;  //convert to colour
                dstImg.setRGB(x, y, clr); //set pixel colout
    }}}
```



Map – image threshholding

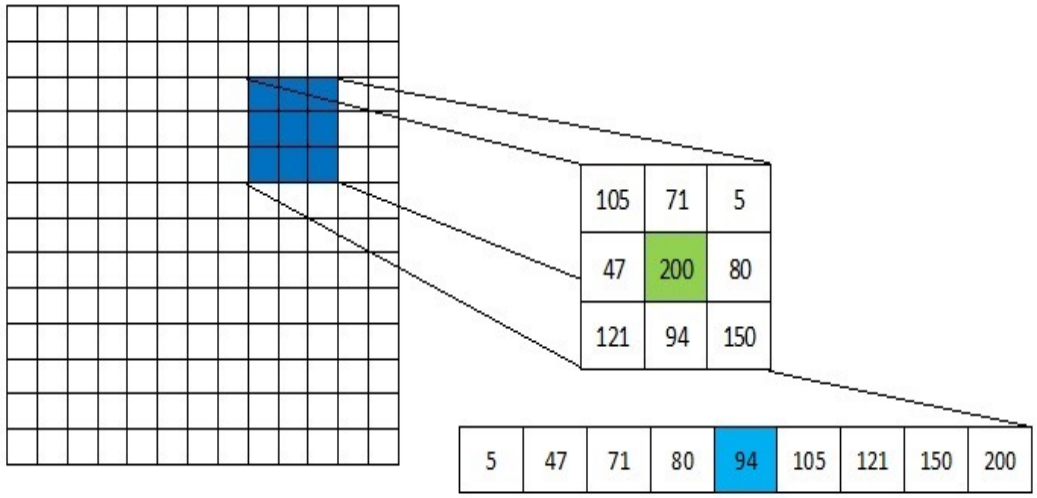
```
    protected void compute() {
        if ((x_stop-x_start)*(y_stop-y_start) < sThreshold) {
            work();
            return;
        }
        int splt = (x_stop-x_start) / 2;
        ParallelTest1 left = new ParallelTest1(img, x_start,x_start+splt, y_start, y_stop,  dstImg);
        ParallelTest1 right = new ParallelTest1(img, x_start+splt, x_stop, y_start, y_stop, dstImg);
                left.fork();
                right.compute();
                left.join();
    }

    public static void main(String[] args) throws Exception {
        File f=new File("Images/image1.jpg");
        BufferedImage img=ImageIO.read(f);
        int w=img.getWidth();
        int h=img.getHeight();
        BufferedImage dstImg = new BufferedImage(w, h, BufferedImage.TYPE_INT_ARGB);

        ForkJoinPool pool = ForkJoinPool.commonPool();
        pool.invoke(new ParallelTest1(img, 0, w, 0, h, dstImg));
        File dstFile = new File("Images/output.png");
        ImageIO.write(dstImg, "png", dstFile);
}}
```

More complex image operation: median smoothing

- A *mean filter* sets each pixel in the image to the *average* of the surrounding pixels, whereas a **median filter** sets each pixel to the **median** of the surrounding pixels.
- A median filter is much better at preserving edges after smoothing (compare the skylines in the three images).



original

mean filter

median filter

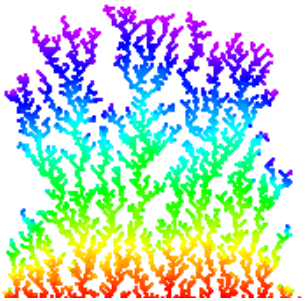
Maps and reductions

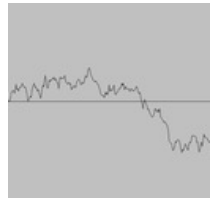
Maps and **reductions**: the "work horses" of parallel programming

- By far the two most important and common **patterns**
- Learn to recognize when an algorithm can be written in terms of maps and reductions
- Use maps and reductions to describe (parallel) algorithms
- Programming them becomes "trivial" with a little practice
  - Exactly like sequential for-loops seem second-nature

Monte Carlo Methods

Basis of Monte Carlo methods is the use of random selections in calculations that lead to the solution of numerical and physical problems e.g.





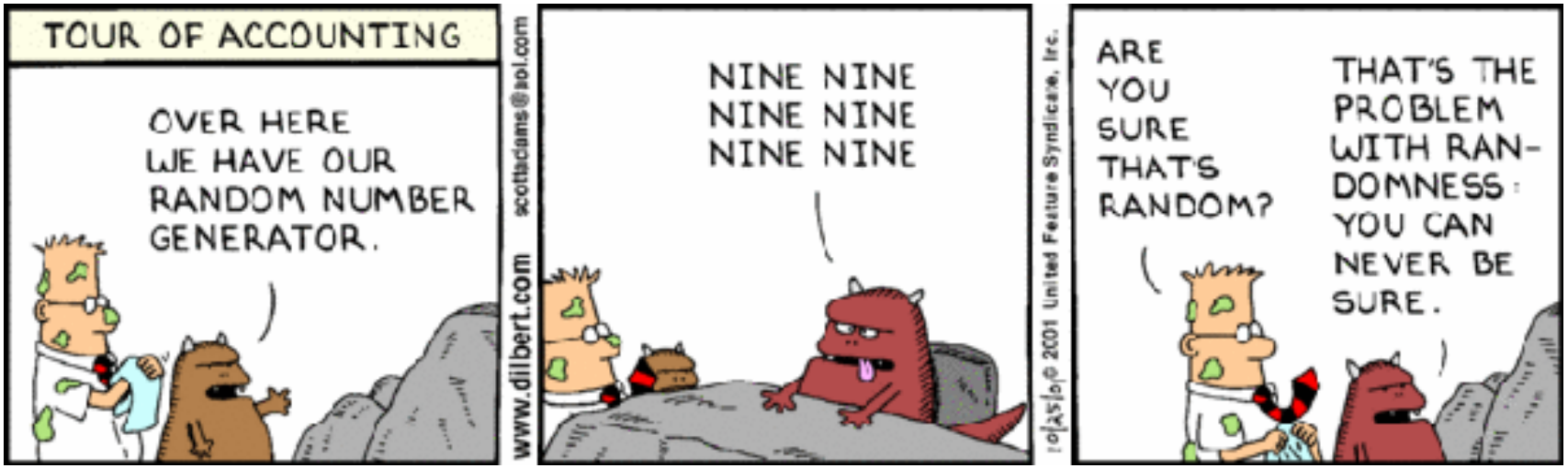
- brownian motion
- molecular modelling
- forecasting the stock market
  - for pricing derivatives, risk analysis, and portfolio optimization
- for **numerical integration** in high-dimensional problems where traditional methods are inefficient or infeasible.

Each calculation is independent of the others and hence **embarrassingly parallel**.

Aside: Parallel Random Number Generation

For successful Monte Carlo simulations, the **random numbers must be independent** of each other

- Developing random number generator algorithms and implementations that are fast, easy to use, and give **good quality pseudo-random numbers** is a challenging problem
  - Developing parallel implementations is even more difficult.



Random gives you pseudorandom numbers

- A **pseudorandom number generator** (**PRNG**) generates a sequence of numbers whose properties approximate a truly random sequence.
- However, the sequence is not truly random, because it is deterministic and completely determined by an initial value, called the **seed**.

```
Random rand = new Random();
```

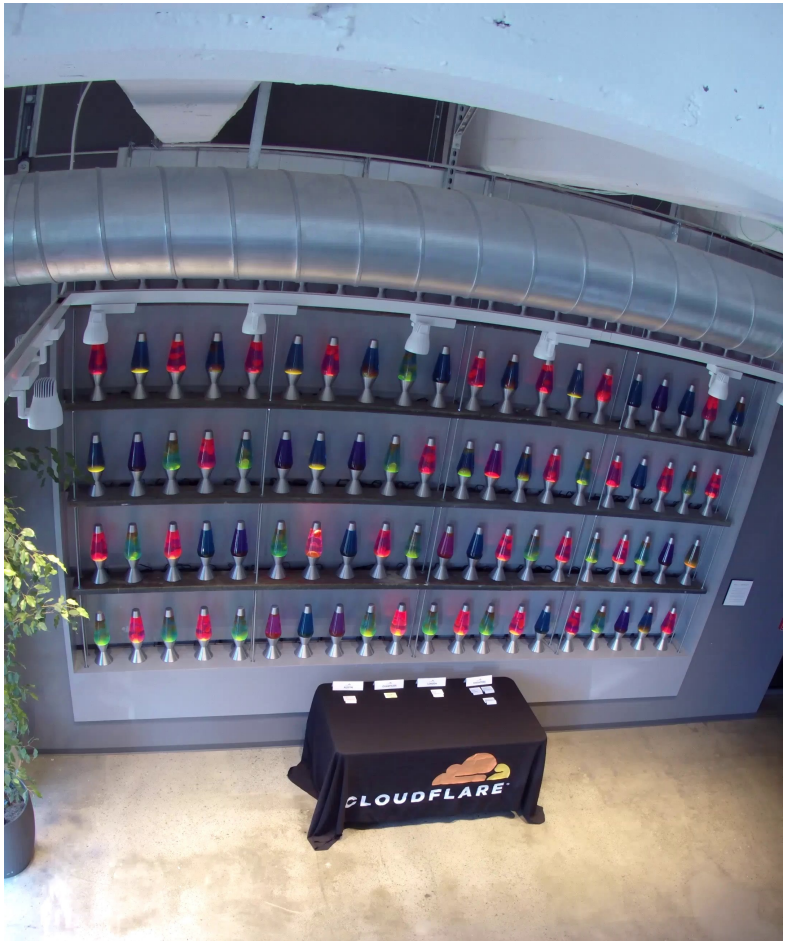
uses nanoTime to try to get a different seed every time an object of Random is created.

Monte Carlo simulations, require **high-quality random numbers** because the reliability of their results depends on the statistical properties of the randomness used. Poor-quality random numbers can lead to **biased, unstable, or misleading results**.

Getting really random numbers

To get really random numbers you need to measure a **truly random physical phenomenon**

- like emission of particles from radioactive decay



- CloudFlare uses lava lamps as a source of randomness (first done as LavaRand in 1996 by Silicon Graphics)
- A wall of lava lamps provides an unpredictable input to a camera aimed at the wall.
- Since the flow of the “lava” in a lava lamp is very unpredictable, “measuring” the lamps by taking footage of them is a good way to obtain unpredictable randomness.
- https://www.cloudflare.com/learning/ssl/lava-lamp-encryption/

Requirements for a Parallel Generator

For random number generators on parallel computers, it is vital that there are **no correlations** between the random number streams on different processors.

- e.g. don't want one processor repeating part of another processor's sequence.
- could occur if we just use the naive method of running a RNG on each different processor and just giving randomly chosen seeds to each processor.

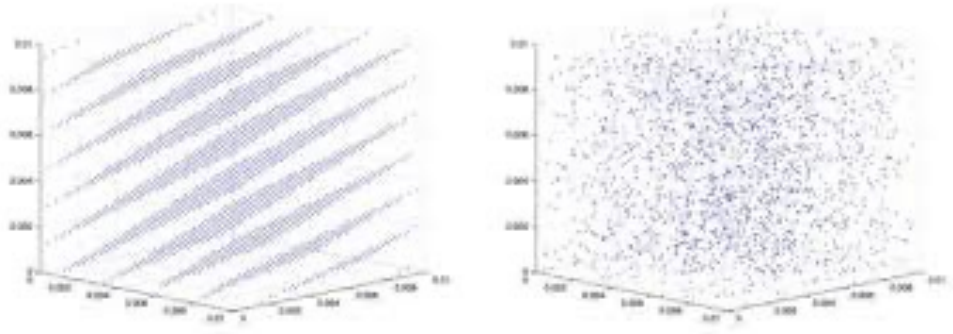
- In many applications we also need to ensure that we get the same results for any number of processors.

Good quality pseudo-random numbers

Random numbers need to be *random enough* for a given purpose

The Mersenne Twister (MT) is a pseudorandom number generator algorithm developed by Matsumoto and Nishimura. It has

- Good distribution properties.
- Long period.
- Efficient use of memory
- High performance.



Default in Python (Julia, Matlab…), not in C or Java.

Makoto Matsumoto, Keio University/Max-Planck_Institut fur Mathematik; Takuji Nishimura, Keio University.
Mersenne Twister. A 623-dimensionally equidistributed uniform pseudorandom number generator.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf

Parallel Mersenne Twister

Look at this if interested

The simplest solution is to have many simultaneous Mersenne twisters processed in parallel. But even “very different” initial state values do not prevent correlated sequences by generators sharing identical parameters.

- dcmt, a special offline library for the dynamic creation of Mersenne Twisters parameters, was developed by Matsumoto and Nishimura
- The library accepts the 16-bit “thread id” as one of the inputs, and encodes this value into the Mersenne Twister parameters on a per-“thread” basis, so that every thread can update the twister independently, while still retaining good randomness of the final output.

Makoto Matsumoto and Takaji Nishimura.
Dynamic Creation of Pseudorandom Number Generators.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dgene.pdf

Java and parallel random numbers

java.util.Random is **thread safe** but can have **poor performance** in a multi-threaded environment due to **contention** when multiple threads share the same *Random* instance.

- prefer ThreadLocalRandom.

I have to use this for my Assignment 1 to generate random numbers

public class **ThreadLocalRandom** extends Random

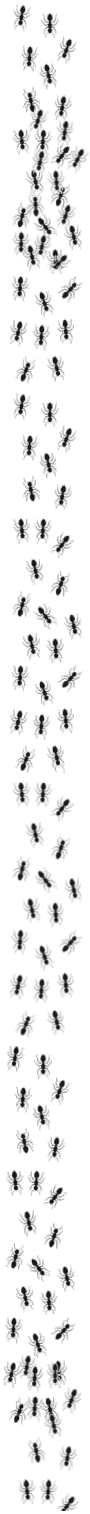
A random number generator isolated to the current thread. Like the global Random generator used by the Math class, a ThreadLocalRandom is initialized with an internally generated seed that may not otherwise be modified. When applicable, use of ThreadLocalRandom rather than shared Random objects in concurrent programs will typically encounter much less overhead and contention. Use of ThreadLocalRandom is particularly appropriate when multiple tasks (for example, each a ForkJoinTask) use random numbers in parallel in thread pools.

Quality of ThreadLocalRandom?

Numbers “appear to be adequate for "everyday" use, such as in Monte Carlo algorithms and randomized data structures where speed is important.”

I have to use this for my assignme

Guy L. Steele, Doug Lea, and Christine H. Flood. 2014. Fast splittable pseudorandom number generators. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14). Association for Computing Machinery, New York, NY, USA, 453–472. https://doi.org/10.1145/2660193.2660195



Interesting parallel example: The prefix-sum problem

Given **`int[] input`**, produce **`int[] output`** where **`output[i]`** is the sum of **`input[0]+input[1]+…+input[i]`**

Sequential can be a CS1 exam problem:

```
int[] prefix_sum(int[] input){
  int[] output = new int[input.length];
  output[0] = input[0];
  for(int i=1; i < input.length; i++)
    output[i] = output[i-1]+input[i];
  return output;
}
```

Does not *seem* parallelizable….

Parallel prefix-sum

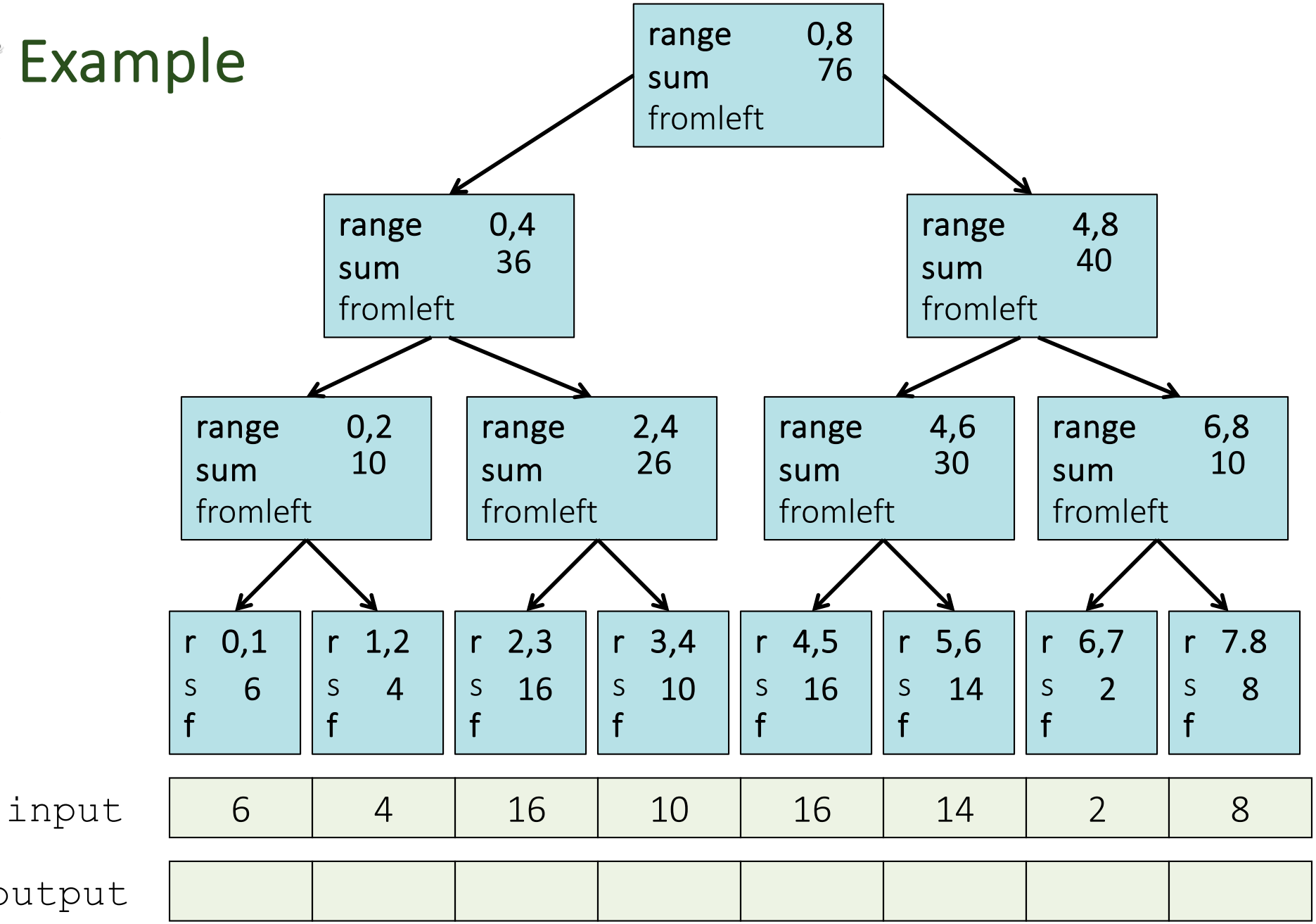
The parallel-prefix algorithm does two passes

- The first pass builds a tree bottom-up: the “up” pass
- The second pass traverses the tree top-down: the “down” pass

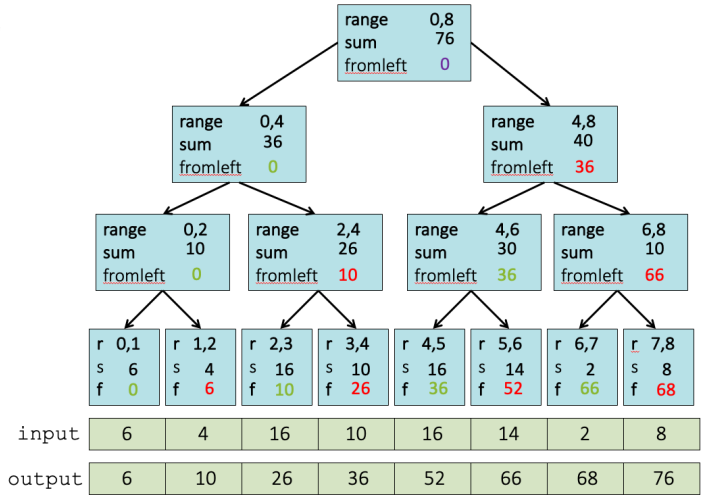
Historical note:

- Original algorithm due to R. Ladner and M. Fischer at the University of Washington in 1977

Example


range 0,8
sum 76
fromleft
range 0,4
sum 36
fromleft
range 4,8
sum 40
fromleft
range 0,2
sum 10
fromleft
range 2,4
sum 26
fromleft
range 4,6
sum 30
fromleft
range 6,8
sum 10
fromleft
r 0,1
s 6
f
r 1,2
s 4
f
r 2,3
s 16
f
r 3,4
s 10
f
r 4,5
s 16
f
r 5,6
s 14
f
r 6,7
s 2
f
r 7.8
s 8
f
input
6
4
16
10
16
14
2
8
output

The algorithm, part 1

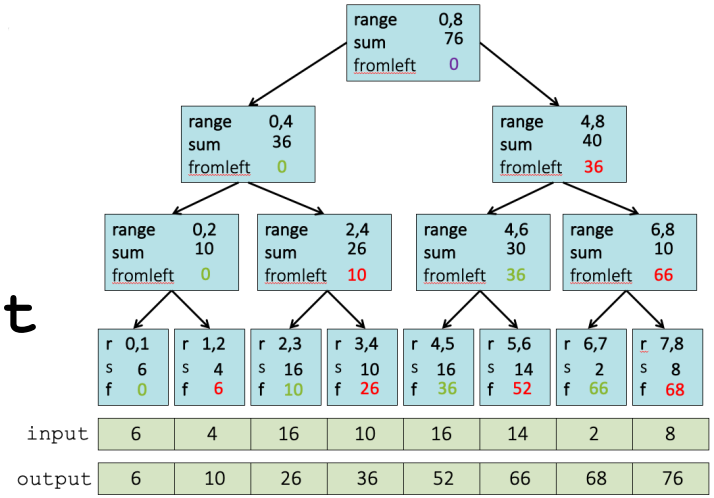


1. **Up:** Build a binary tree where
   - Root has sum of the range [**`x,y`**)
   - If a node has sum of [**`lo,hi`**) and **`hi>lo`**,
     - Left child has sum of [**`lo,middle`**)
     - Right child has sum of [**`middle,hi`**)
     - A leaf has sum of [**`i,i+1`**), i.e., **`input[i]`**

This is an easy fork-join computation: combine results by actually building a binary tree with all the range-sums

- Tree built bottom-up in parallel
- Could be more clever with an array, as with heaps

The algorithm, part 2

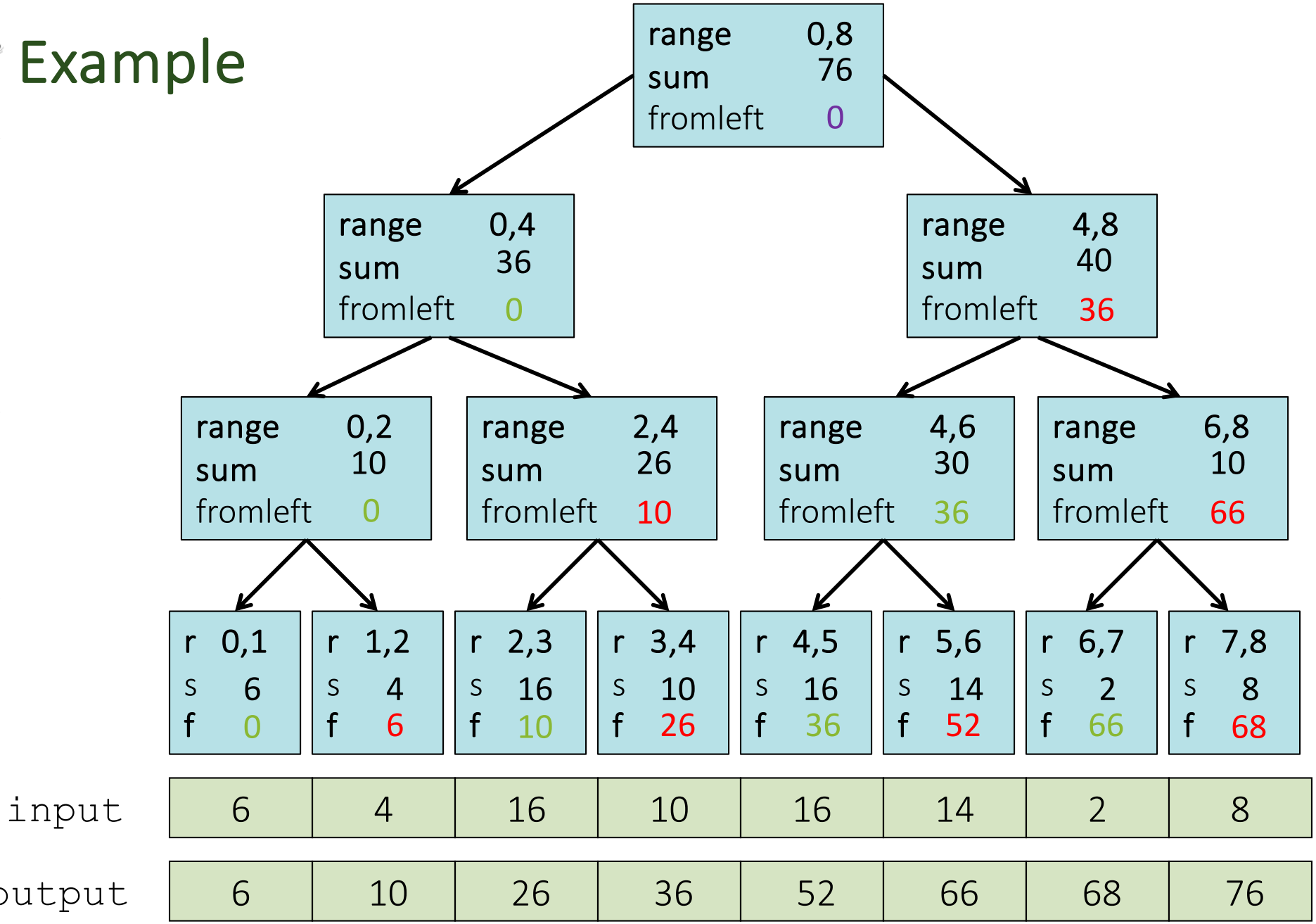


2. **Down:** Pass down a value **`fromLeft`**
   - Root given a **`fromLeft`** of **`0`**
   - Node takes its **`fromLeft`** value and
     - Passes its left child the same **`fromLeft`**
     - Passes its right child its **`fromLeft`** plus its left child's **`sum`** (as stored in part 1)
   - At the leaf for array position **`i`**, **`output[i]=fromLeft+input[i]`**

This is an easy fork-join computation: traverse the tree built in step 1 and produce no result

- Leaves assign to **`output`**
- Invariant: **`fromLeft`** is sum of elements left of the node's range

Example


range 0,8
sum 76
fromleft 0
range 0,4
sum 36
fromleft 0
range 4,8
sum 40
fromleft 36
range 0,2
sum 10
fromleft 0
range 2,4
sum 26
fromleft 10
range 4,6
sum 30
fromleft 36
range 6,8
sum 10
fromleft 66
r 0,1
s 6
f 0
r 1,2
s 4
f 6
r 2,3
s 16
f 10
r 3,4
s 10
f 26
r 4,5
s 16
f 36
r 5,6
s 14
f 52
r 6,7
s 2
f 66
r 7,8
s 8
f 68
input
6
4
16
10
16
14
2
8
output
6
10
26
36
52
66
68
76

Sequential cut-off

Adding a sequential cut-off is easy as always:

- Up:

  just a sum, have leaf node hold the sum of a range

- Down:

```
output[lo] = fromLeft + input[lo];
 for(i=lo+1; i < hi; i++)
    output[i] = output[i-1] + input[i]
```

Parallel prefix, generalized

Just as sum-array was the simplest example of a pattern that matches many, many problems, so is prefix-sum

- Minimum, maximum of all elements to the left of `i`
- Is there an element to the left of **i** satisfying some property?
- Count of elements to the left of **i** satisfying some property
  - This last one is perfect for an efficient parallel pack…
  - Perfect for building on top of the "parallel prefix trick"
- We did an *inclusive* sum, but *exclusive* is just as easy

Pack

[Non-standard terminology]

Given an array **`input`**, produce an array **`output`** containing only elements such that **`f(x)`** is **`true`**

Example: **`input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`**

`f: x > 10`

`output [17, 11, 13, 19, 24]`

Parallelizable?

- Finding elements for the output is easy
- But getting them in the right place seems hard

Parallel prefix to the rescue

1. Parallel map to compute a bit-vector for true elements

```
input  [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits   [1,  0, 0, 0,  1, 0,  1,  1, 0,  1]
```

2. Parallel-**prefix sum** on the bit-vector

```
bitsum [1,  1, 1, 1,  2, 2,  3,  4, 4,  5]
```

3. Parallel **map** to produce the output

```
output [17, 11, 13, 19, 24]
```

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
  if(bits[i]==1)
    output[bitsum[i]-1] = input[i];
}
```

Pack comments

First two steps can be combined into one pass

- Just using a different base case for the prefix sum

- Can also combine third step into the down pass of the prefix sum

- Parallelized packs can help us parallelize quicksort…

Quicksort review

- Very popular sequential sorting algorithm that performs well with an average sequential time complexity of O(*n*log*n*).
  - First list divided into two sublists.
    - All the numbers in one sublist arranged to be smaller than all the numbers in the other sublist.
- Achieved by first selecting one number, called a *pivot*, against which every other number is compared.
  - If the number is less than the pivot, it is placed in one sublist. Otherwise, it is placed in the other sublist.

Quicksort review

Recall quicksort was sequential, in-place, expected time $O(n \log n)$

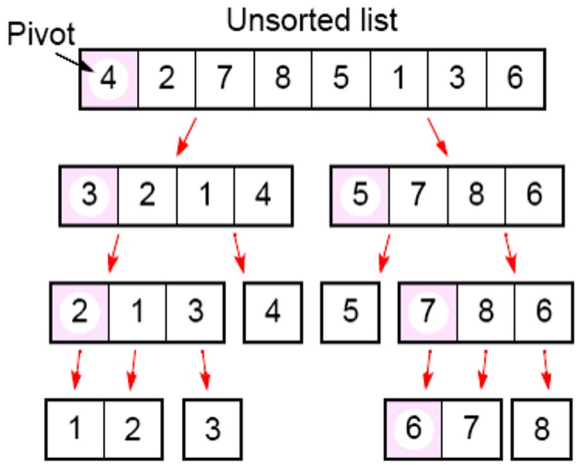
1. Pick a pivot element
2. Partition all the data into:
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than the pivot
3. Recursively sort A and C

How should we parallelize this?

Quicksort

1. Pick a pivot element
2. Partition all the data into:
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than the pivot
3. Recursively sort A and C

Easy: Do the two recursive calls in parallel



Doing better

Can also parallelize the partition

- But we need auxiliary storage (no longer in place)
- In practice, constant factors may make it not worth it

- Already have everything we need to do this…

Parallel partition (not in place)

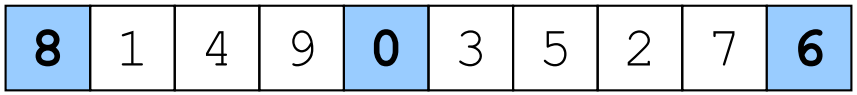
Partition all the data into:

A. The elements less than the pivot
B. The pivot
C. The elements greater than the pivot

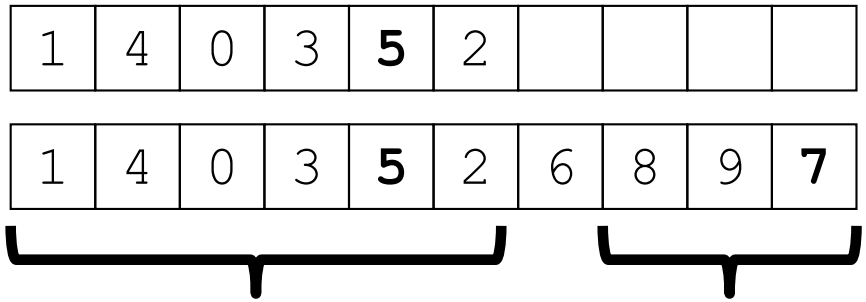
- This is just two packs!
  - Pack elements less than pivot into left side of **`aux`** array
  - Pack elements greater than pivot into right size of **`aux`** array
  - Put pivot between them and recursively sort
  - With a little more cleverness, can do both packs at once but no effect on asymptotic complexity

Example

- Step 1: pick pivot as median of three



- Steps 2a and 2c (combinable): pack less than, then pack greater than into a second array
  - Fancy parallel prefix to pull this off not shown (for assignment)



- Step 3: Two recursive sorts in parallel
  - Can sort back into original array (like in mergesort)