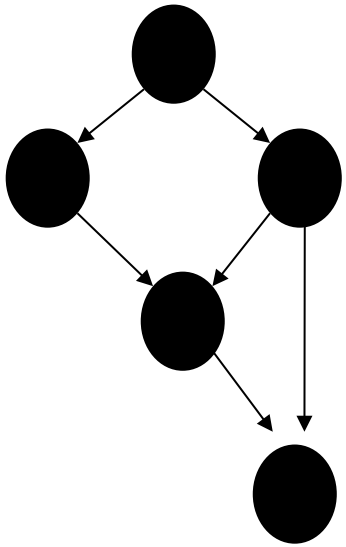
Section 4: Parallel Algorithms

Michelle Kuttel
mkuttel@cs.uct.ac.za

The DAG, or "cost graph"

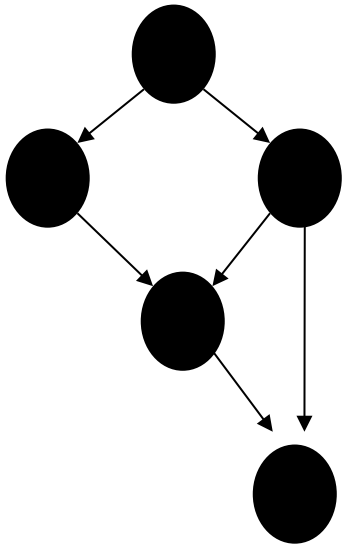
- A program execution using **`fork`** and **`join`** can be seen as a DAG (directed acyclic graph)
  - Nodes: Pieces of work
  - Edges: Source must finish before destination starts



- A `fork` "ends a node" and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A `join` "ends a node" and makes a node with two incoming edges
  - Node just ended
  - Last node of thread joined on

The DAG, or “cost graph”

- work – number of nodes
- span – length of the longest path
  - critical path



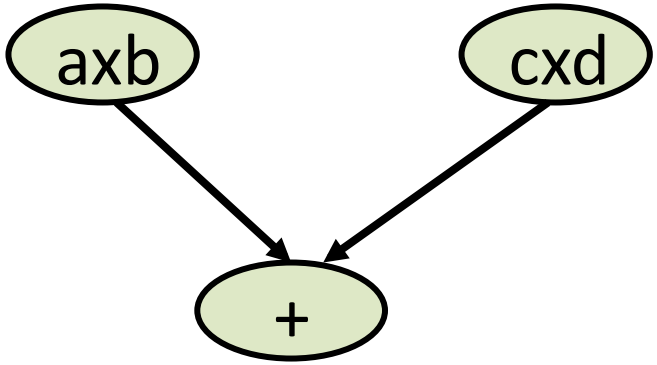
Checkpoint:
What is the span of this DAG?
What is the work?

Checkpoint

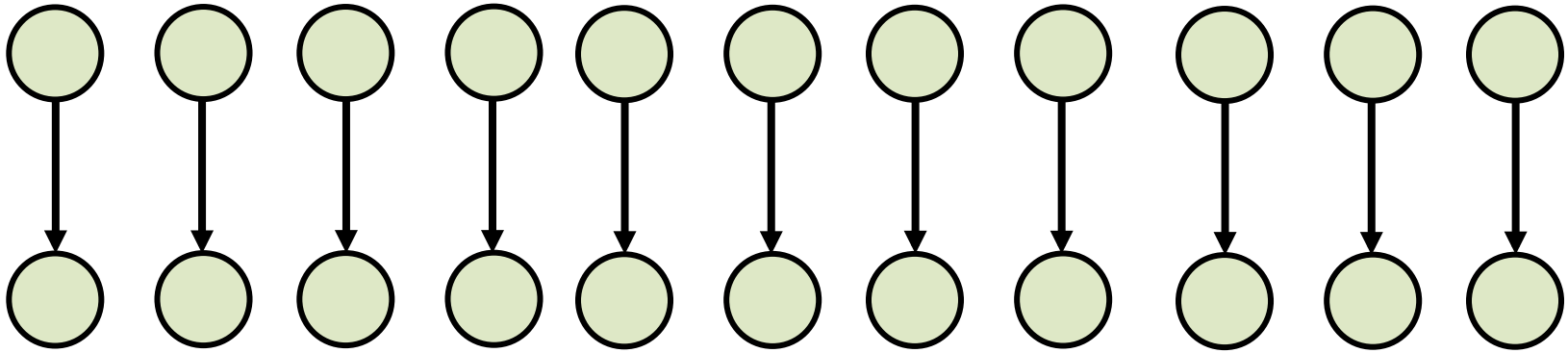
axb + cxd

- Write a DAG to show the the *work* and *span* of this expression

•the set of instructions forms the vertices of the dag
• the graph edges indicate dependences between instructions.
•We say that an instruction x ***precedes an instruction y if x must complete before y can begin.***



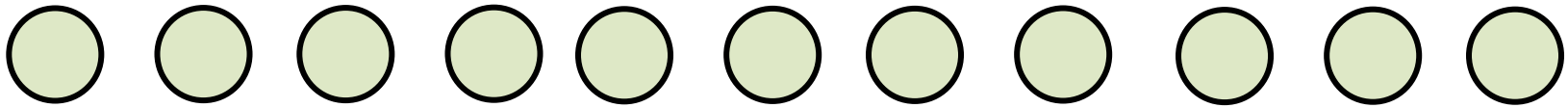
DAG for an embarrassingly parallel algorithm



$$y_i = f_i(x_i)$$

DAG for an embarrassingly parallel algorithm

or, indeed:



$$y_i = f_i(x_i)$$

Embarrassingly parallel examples

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

No special algorithms or techniques required to get a workable solution e.g.

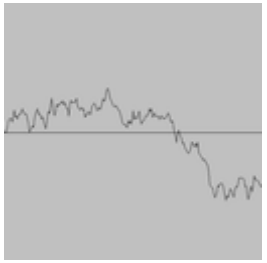
- element-wise linear algebra:
  - addition, scalar multiplication etc
- Image processing
  - shift, rotate, clip, scale
- Monte Carlo simulations
- encryption, compression

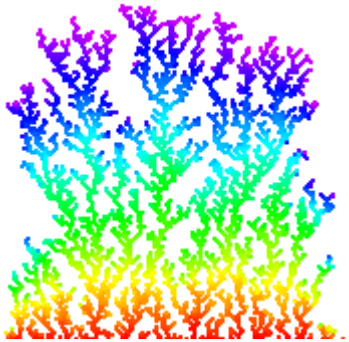
Image Processing

- 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
    - well known Mandelbrot set
  - local operations – produce output based on a group of neighbouring pixels
  - global operations – produce output based on all the pixels of the image
- Point processing operations are embarrassingly parallel (local operations are often highly parallelizable)

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

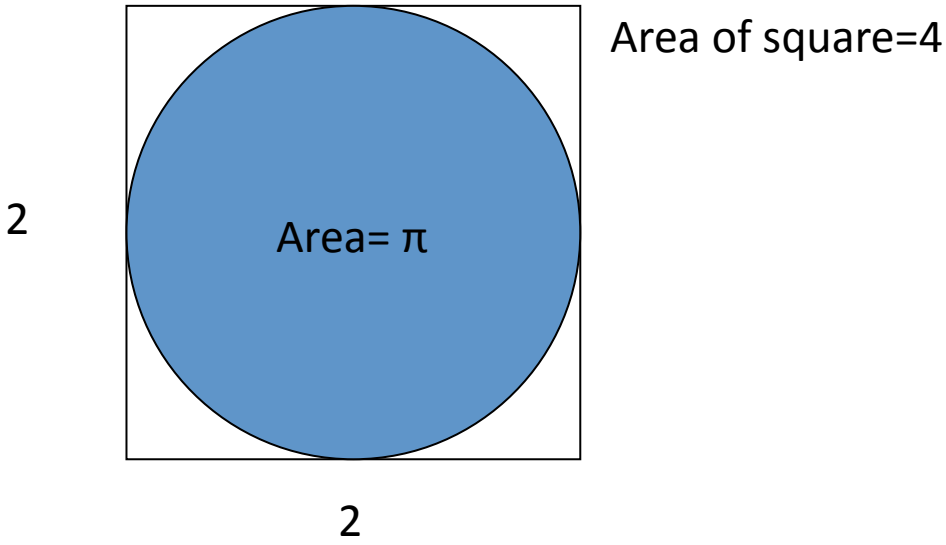




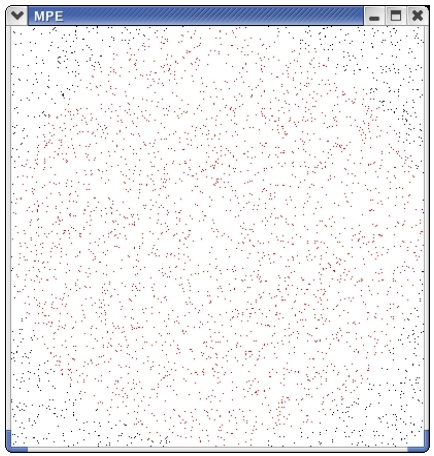
- Each calculation is independent of the others and hence amenable to embarrassingly parallel methods

Trivial Monte Carlo Integration : finding value of π

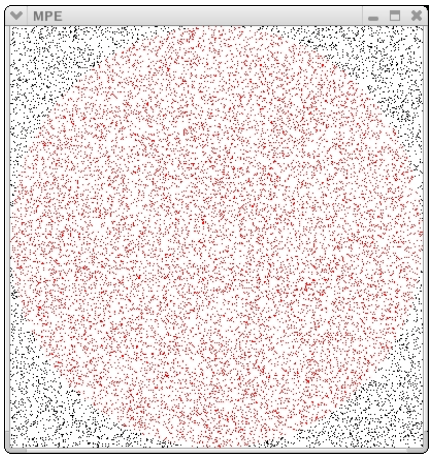
- Monte Carlo integration
  - Compute r by generating **random points** in a square of side 2 and counting how many of them are in the circle with radius 1 ($x^2+y^2<1$; π=4**ratio*) .



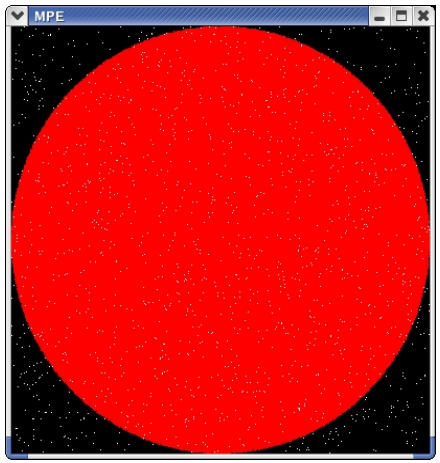
Monte Carlo Integration : finding value of π



0.001



0.0001



0.00001

solution visualization

Monte Carlo Integration

- Monte Carlo integration can also be used to calculate
  - the area of any shape within a known bound area
  - any area under a curve
  - any definite integral
- Widely applicable brute force solution.
  - Typically, accuracy is proportional to square root of number of repetitions.
- Unfortunately, Monte Carlo integration is very computationally intensive, so used when other techniques fail.
- also requires the maximum and minimum of any function within the region of interest.

Note: 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.

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.

Parallel Random Numbers

- three general approaches to the generation of random numbers on parallel computers:
  - *centralized approach*
    - a sequential generator is encapsulated in a task from which other tasks request random numbers. This avoids the problem of generating multiple independent random sequences, but is unlikely to provide good performance. Furthermore, it makes reproducibility hard to achieve: the response to a request depends on when it arrives at the generator, and hence the result computed by a program can vary from one run to the next

Parallel Random Numbers

– *replicated approach:*

- multiple instances of the same generator are created (for example, one per task).
- Each generator uses either the same seed or a unique seed, derived, for example, from a task identifier.
- Clearly, sequences generated in this fashion are not guaranteed to be independent and, indeed, can suffer from serious correlation problems. However, the approach has the advantages of efficiency and ease of implementation and should be used when appropriate.

Parallel Random Numbers

- *distributed approach:*
- responsibility for generating a single sequence is partitioned among many generators, which can then be parceled out to different tasks. The generators are all derived from a single generator; hence, the analysis of the statistical properties of the distributed generator is simplified.

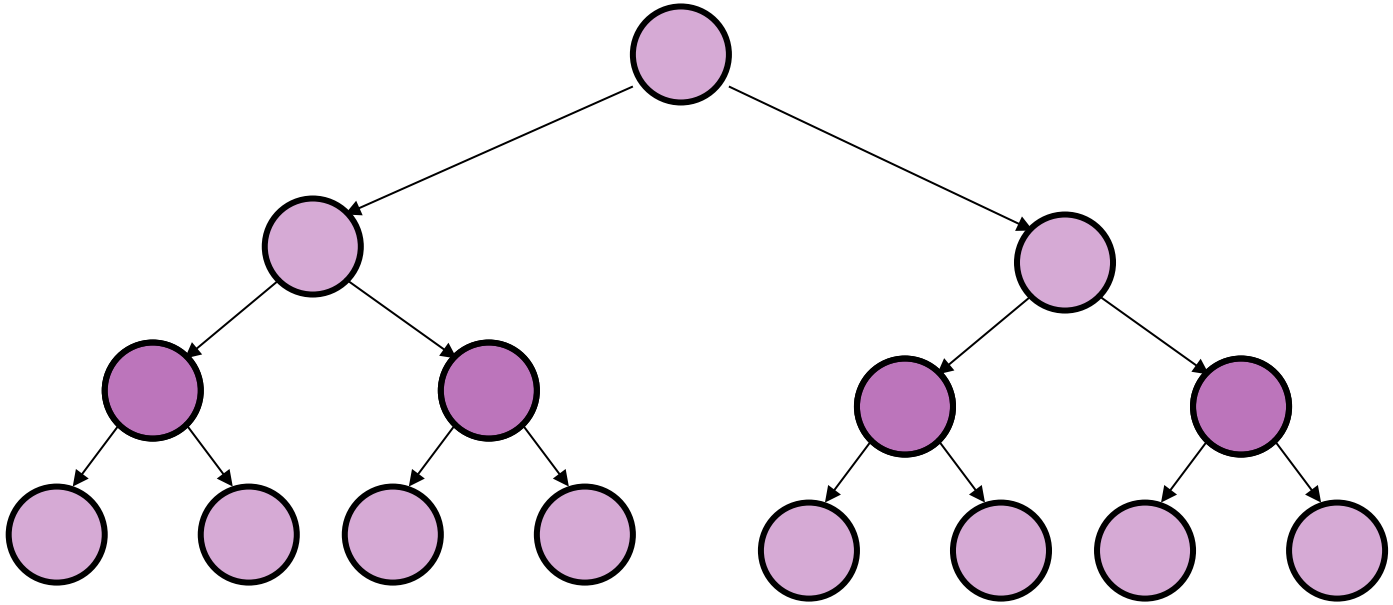
Divide-and-conquer algorithms

- characterized by dividing 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.

Divide-and-conquer algorithms

- binary tree if 2 parts at each division
  - traversed down when calls are made
  - up when calls return

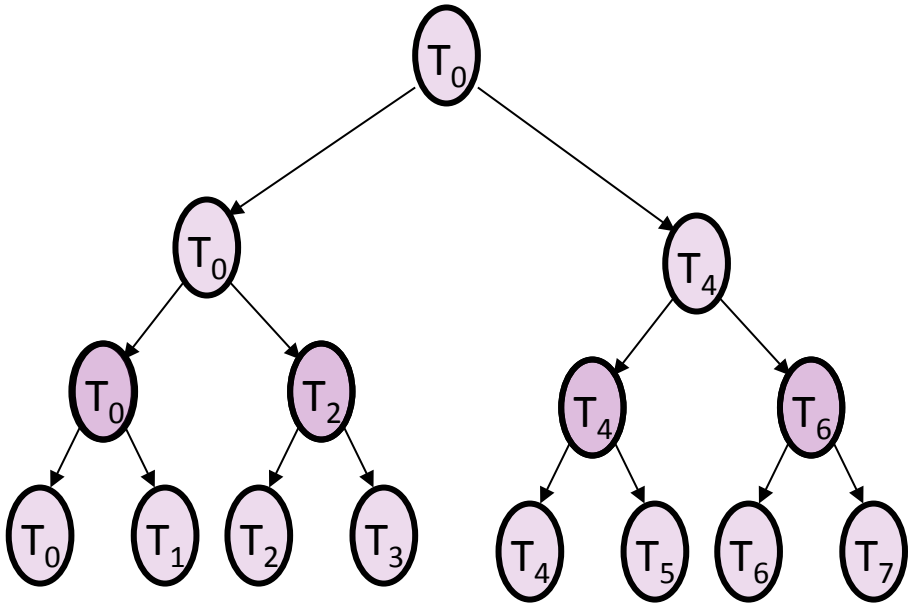


Parallel implementations of Divide-and-conquer

- Sequential implementation can only visit **one node at a time**
- Parallel implementation can traverse several parts of the tree **simultaneously**
- could assign one thread to each node in the tree
  - 2^{m+1}-1 processors in 2^m parts
  - inefficient solution
    - Each processor only active at one level of the tree

Divide-and-conquer – Parallel implementation

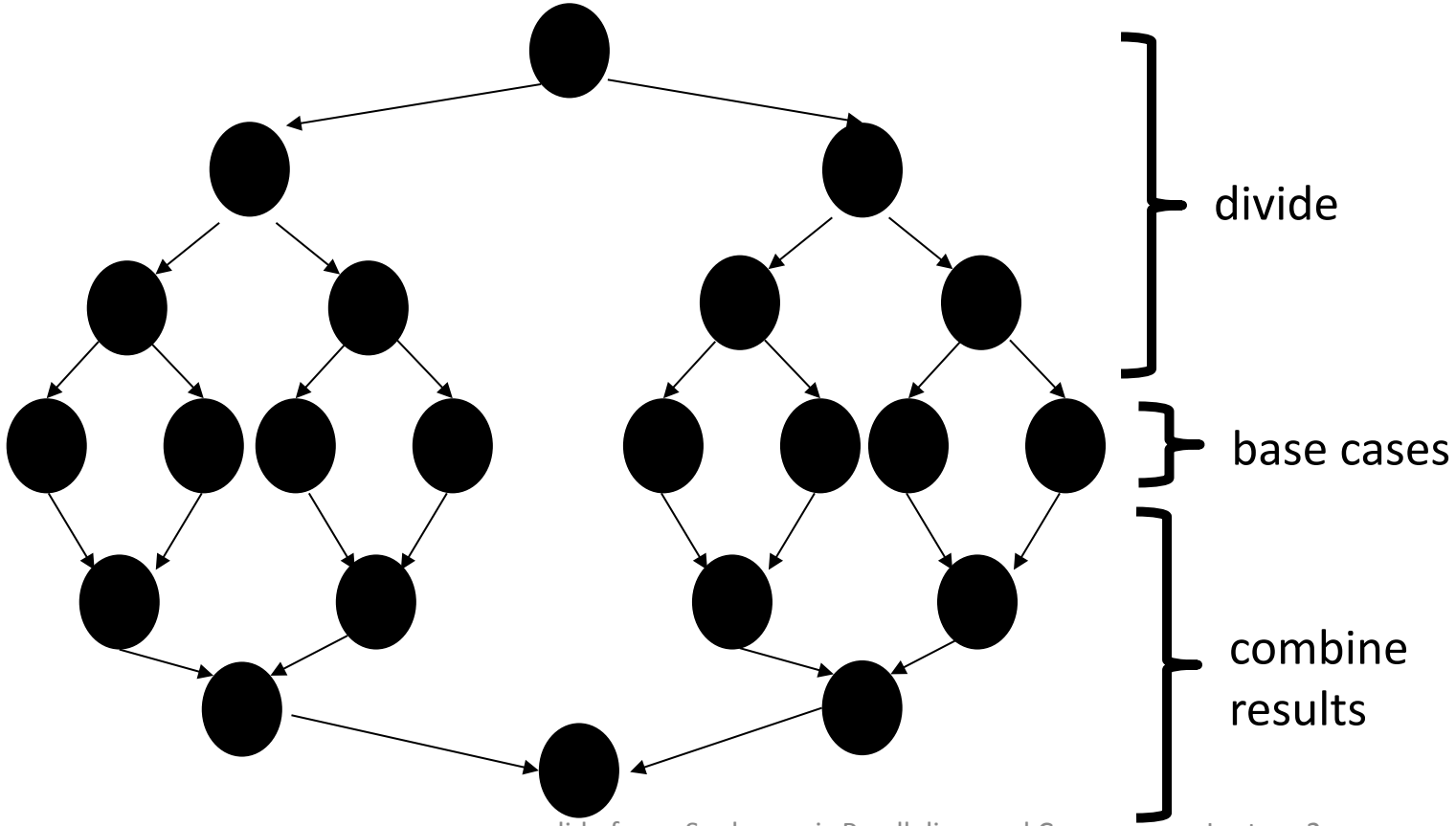
- more efficient: reuse thread at each level of the tree
  - at each stage, thread keeps half the list and passes on the other half
  - each list will have n/t numbers



- summing an array went from $O(n)$ sequential to $O(\log n)$ parallel
- An **exponential speed-up** in theory (*assuming **a lot** of processors and very large n!*)

Our simple examples

- **`fork`** and **`join`** are very flexible, but divide-and-conquer maps and reductions use them in a very basic way:
  - A tree on top of an upside-down tree



Connecting to performance

- Recall: $\mathbf{T_P}$ = running time if there are **P** processors available
- Work = $\mathbf{T_1}$ = sum of run-time of all nodes in the DAG
  - That lonely processor does everything
  - Any topological sort is a legal execution
  - $O(n)$ for simple maps and reductions
- Span = $\mathbf{T_\infty}$ = sum of run-time of all nodes on the most-expensive path in the DAG
  - Note: costs are on the nodes not the edges
  - Our infinite army can do everything that is ready to be done, but still has to wait for earlier results
  - $O(\log n)$ for simple maps and reductions

Optimal T_P: Thanks ForkJoin library!

- So we know $\mathbf{T_1}$ and $\mathbf{T_\infty}$ but we want $\mathbf{T_P}$ (e.g., **P**=4)
- Ignoring memory-hierarchy issues (caching), $\mathbf{T_P}$ can't beat
  - $\mathbf{T_1 / P}$ *why not?*
  - $\mathbf{T_\infty}$ *why not?*
- So an *asymptotically* optimal execution would be:

$$\mathbf{T_P = O((T_1 / P) + T_\infty)}$$

  - First term dominates for small **P**, second for large **P**
- The ForkJoin Framework gives an *expected-time guarantee* of asymptotically optimal!
  - Expected time because it flips coins when *scheduling*
  - How? For an advanced course (few need to know)
  - Guarantee requires a few assumptions about your code…

Definition

- An algorithm is said to be asymptotically optimal if, for large inputs, it performs at worst a constant factor (independent of the input size) worse than the best possible algorithm.

What that means (mostly good news)

The fork-join framework guarantee

$$T_P \leq (T_1 / P) + O(T_\infty)$$

- No implementation of your algorithm can beat $O(T_\infty)$ by more than a constant factor
- No implementation of your algorithm on P processors can beat (T_1 / P) (ignoring memory-hierarchy issues)
- So the framework on average gets within a constant factor of the best you can do, assuming the user (you) did his/her job

So: You can focus on your algorithm, data structures, and cut-offs rather than number of processors and scheduling

- Analyze running time given T_1, T_∞, and P

Division of responsibility

- Our job as ForkJoin Framework users:
  - Pick a good algorithm
  - Write a program. When run, it creates a DAG of things to do
  - *Make all the nodes a small-ish and approximately equal amount of work*

- The framework-writer's job (won't study how this is done):
  - Assign work to available processors to avoid idling
  - Keep constant factors low
  - Give the expected-time optimal guarantee assuming framework-user did his/her job

$$T_P = O((T_1 / P) + T_\infty)$$

slide
slide from: Sophomoric Parallelism and Concurrency, Lecture 2

Examples

$$T_P = O((T_1 / P) + T_\infty)$$

- In the algorithms seen so far (e.g., sum an array):
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
  - So expect (ignoring overheads): $T_P = O(n/P + \log n)$

- Suppose instead:
  - $T_1 = O(n^2)$
  - $T_\infty = O(n)$
  - So expect (ignoring overheads): $T_P = O(n^2/P + n)$

Basic 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 entirely on the result of processing `arr[i-1]`

Basic algorithms: Maps (Data Parallelism)

- 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

```
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;
}
```

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;
}
```

Maps and reductions

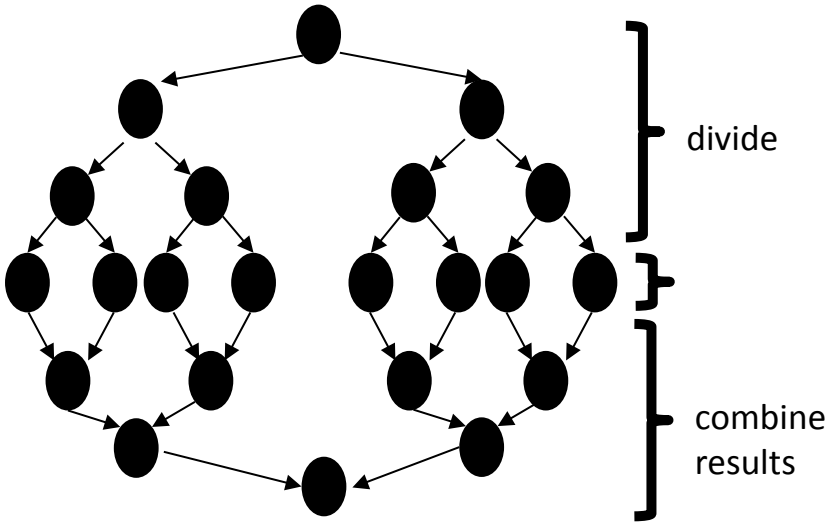
Maps and reductions: the "workhorses" of parallel programming

- By far the two most important and common patterns
  - Two more-advanced patterns in next lecture
- 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

Other examples of divide and conquer

- 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, for example, number of strings that start with a vowel
  - This is just summing with a different base case
  - Many problems are!

More interesting DAGs?



- The DAGs are not always this simple
- Example:
  - Suppose combining two results might be expensive enough that we want to parallelize each one
  - Then each node in the inverted tree on the previous slide would itself expand into another set of nodes for that parallel computation

Next

Clever ways to parallelize more than is intuitively possible

- Parallel prefix:
  - This "key trick" typically underlies surprising parallelization
  - Enables other things like packs
- Parallel sorting: quicksort (not in place) and mergesort
  - Easy to get a little parallelism
  - With cleverness can get a lot