Unit #8: Shared-Memory Parallelism and Concurrency

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

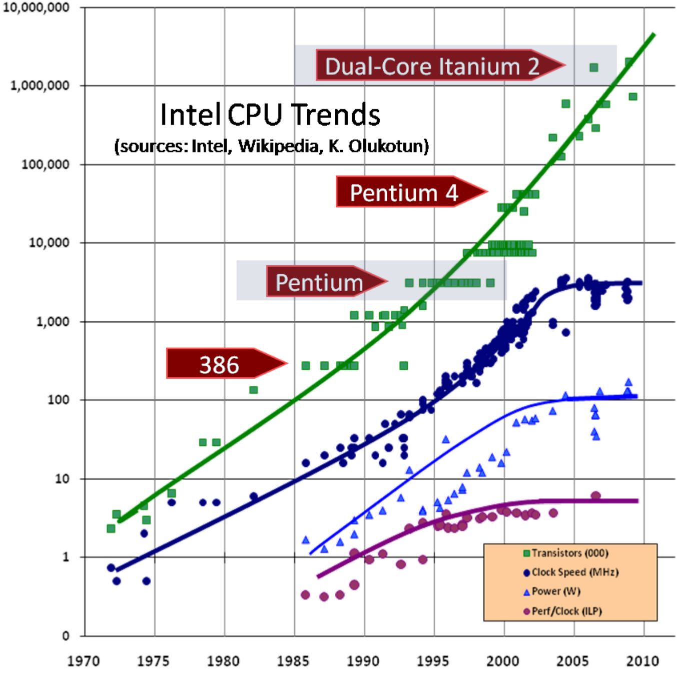
2016W2

Unit Outline

- History and Motivation
- Parallelism versus Concurrency
- Counting Matches in Parallel
- Divide and Conquer
- Reduce and Map
- Analyzing Parallel Programs
- Parallel Prefix Sum

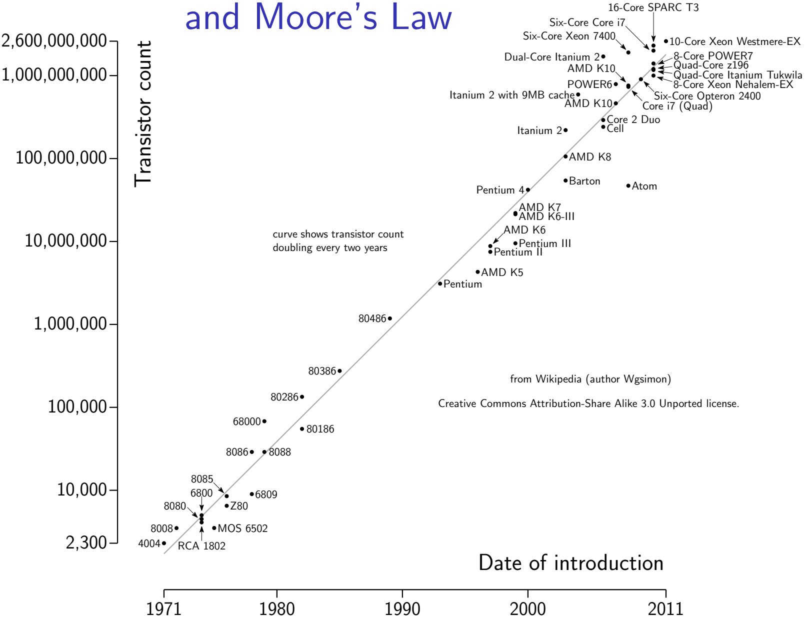
Learning Goals

- Distinguish between **parallelism** – improving performance by exploiting multiple processors – and **concurrency** – managing simultaneous access to shared resources.
- Use the **fork**/**join** mechanism to create parallel programs.
- Represent a parallel program as a **DAG**.
- Define **Work** – the time it takes one processor to complete a computation; **Span** – the time it takes an infinite number of processors to complete a computation; **Amdahl's Law** – the speedup obtainable by parallelizing as a function of the proportion of the computation that is parallelizable.
- Use **Work**, **Span**, and **Amdahl's Law** to analyze the possible speedup of a parallel version of a computation.
- Determine when and how to use parallel **Map**, **Reduce**, and **Prefix Sum** patterns.



The Free Lunch Is Over A Fundamental Turn Toward Concurrency in Software By Herb Sutter

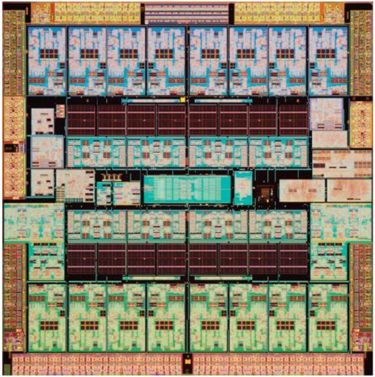
Microprocessor Transistor Counts (1971-2011) and Moore's Law



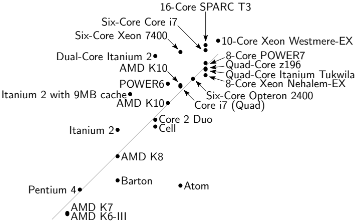
curve shows transistor count doubling every two years

from Wikipedia (author Wgsimon)

Microprocessor Transistor Counts (2000-2011)



16-Core SPARC T3 (Oracle)



from Wikipedia (author Wgsimon)

Parallelism versus Concurrency

Parallelism

Performing multiple steps at the same time.
16 chefs using 16 ovens.

Concurrency

Managing access by multiple agents to a shared resource.
16 chefs using 1 oven.

Who's Doing the Work?

Processor/Core Machine that executes instructions – one instruction at a time.

In reality, each core may execute (parts of) many instructions at the same time.

Process Executing instance of a program.

The operating system schedules when a process executes on a core.

Thread Light-weight process.

Each process may create many threads, but threads are still scheduled by the operating system.

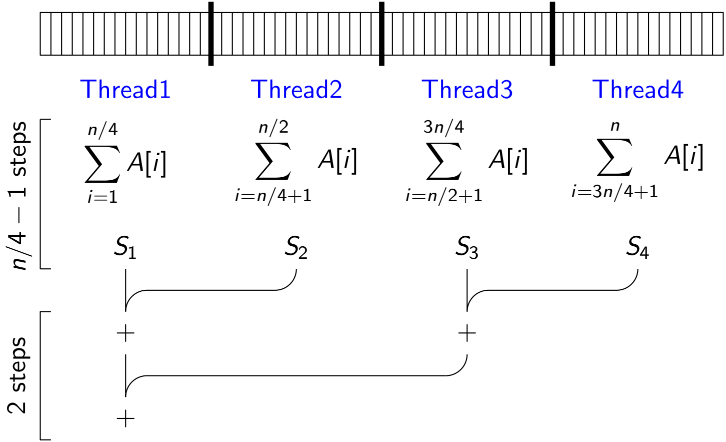
Task Light-weight thread. (in OpenMP 3.x)

A task may be scheduled for execution using a different mechanism than the operating system.

Parallelism

Performing multiple (computation) steps at the same time.

Sum n integers using four processors



Thread1 Thread2 Thread3 Thread4

$$\sum_{i=1}^{n/4} A[i] \qquad \sum_{i=n/4+1}^{n/2} A[i] \qquad \sum_{i=n/2+1}^{3n/4} A[i] \qquad \sum_{i=3n/4+1}^{n} A[i]$$

S_1 S_2 S_3 S_4

$+$ $+$

$+$

$n/4 - 1$ steps

2 steps

Total time: $n/4 + 1$

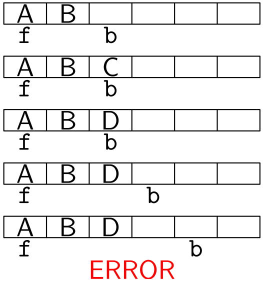
Concurrency

Managing access by multiple executing agents to a shared resource.

Shared Queue

```
void enQ(Obj x) {
  Q[b] = x;
  b=(b+1)%n;
}
```

time	Thread1	Thread2
0	`enQ("C")`	`enQ("D")`
1	`Q[b]="C"`	
2		`Q[b]="D"`
3	`b=(b+1)%n`	
4		`b=(b+1)%n`



Models of Parallel Computation

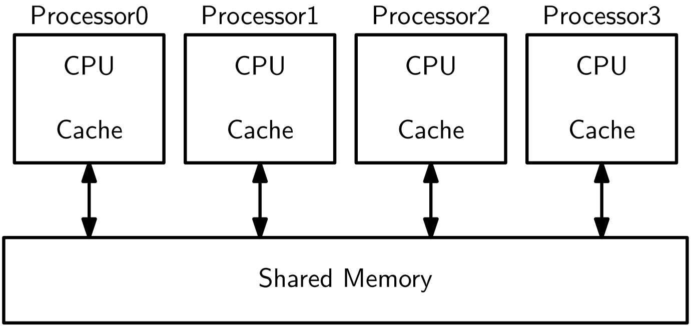
Shared Memory Agents read from and write to a common memory.

Message Passing Agents explicitly send and receive data to/from other agents. (Distributed computing.)

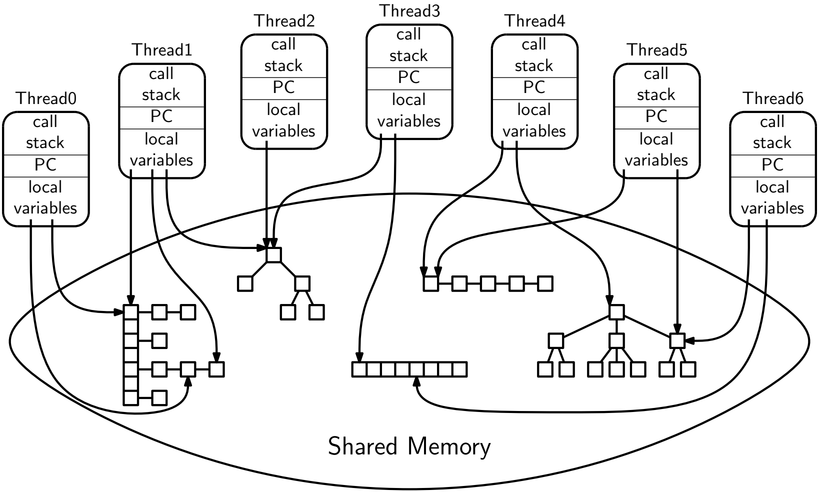
Data flow Agents are nodes in a directed acyclic graph. Edges represent data that an agent needs as input (incoming) and produces as output (outgoing). When all input is available, the agent can produce output.

Data parallelism Certain operations (e.g., sum) execute in parallel on collections (e.g., arrays) of data.

Shared Memory in Hardware



Shared Memory in Software



PC = program counter = address of currently executing instruction

Counting Matches

How many times does the number 3 appear?

3	5	9	3	4	6	7	2	1	8	3	3	5	2	3	9

```
// Sequential version
int nMatches(int A[], int lo, int hi, int key) {
  int m = 0;
  for( int i=lo; i<hi; i++ )
    if( A[i] == key ) m++;
  return m;
}
```

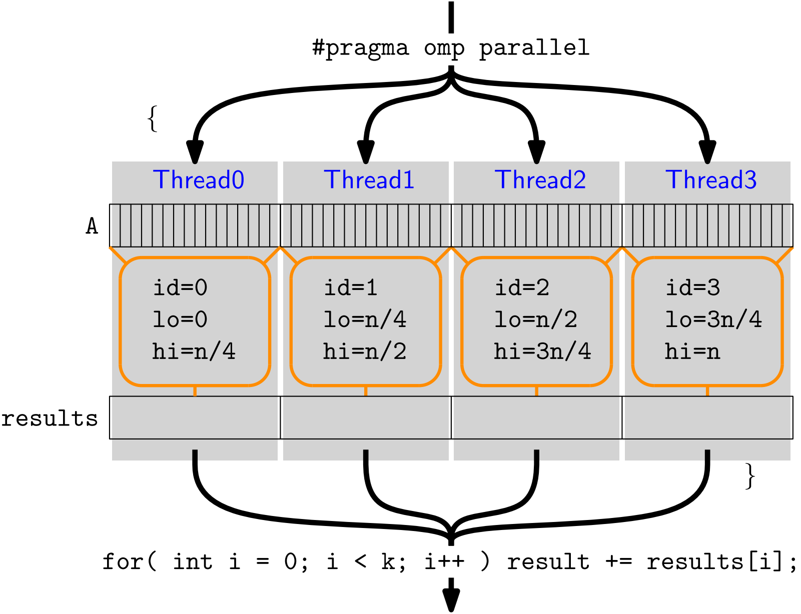
Counting Matches in Parallel

```
#include "omp.h"
int nmParallel(int A[], int n, int key) {
  int k = 4;               if( k > n ) k = n;
  int results[k];          int nn = n/k;
  omp_set_num_threads(k);

  #pragma omp parallel
  {
    int id=omp_get_thread_num(), lo = id * nn, hi;
    if (id == k-1) hi = n; else hi = lo + nn;
    results[id] = nMatches(A, lo, hi, key);
  }
  int result = 0;
  for( int i = 0; i < k; i++ ) result += results[i];
  return result;
}
```

k is the number of threads.

Counting Matches in Parallel (cont.)



How Many Agents (Threads)?

Let n be the array size and k be the number of threads.

1. Divide array into k pieces.
2. Solve these pieces in parallel; time is $\Theta(n/k)$ using k processors.
3. Combine the pieces by summing the results; time is $\Theta(k)$.

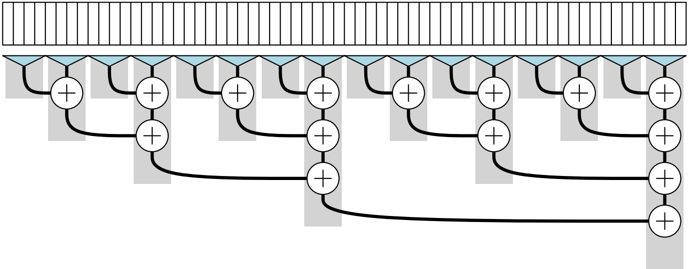
Total time: $\Theta(n/k) + \Theta(k)$

What's the best value of k? $\sqrt{n}$

Couldn't we do better if we had more processors?
Combine is the bottleneck...

Combine in Parallel)

The process of producing a single answer[1] from a list is called **Reduce**.



Reduce using $\oplus$ can be done in parallel, as shown, if

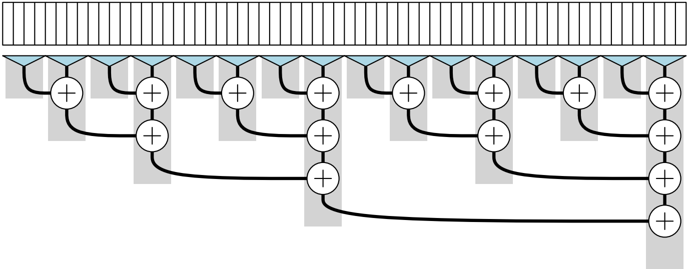
$$((a \oplus b) \oplus c) \oplus d = (a \oplus b) \oplus (c \oplus d)$$

which is true for associative operations.

[1] A "single" answer may be a list or collection of values.

Combine in Parallel (cont.)

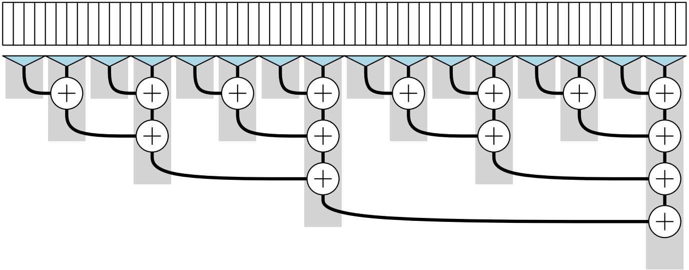
How do we create threads that know how to combine in parallel?



Does this look like anything we've seen before?

Combine in Parallel (cont.)

How do we create threads that know how to combine in parallel?



Does this look like anything we've seen before?

The “merge” part of Mergesort!

Counting Matches with Divide and Conquer

```
int nmpDC(int A[], int lo, int hi, int key) {
  if( hi - lo <= CUTOFF ) return nMatches(A, lo, hi, key);
  int left, right;
#pragma omp task untied shared(left)
  { left = nmpDC(A, lo, (lo + hi)/2, key); }
  right = nmpDC(A, (lo + hi)/2, hi, key);
#pragma omp taskwait
  return left + right;
}
int nmpDivConq(int A[], int n, int key) {
  int result;
#pragma omp parallel
#pragma omp single
  { result = nmpDC(A, 0, n, key); }
  return result;
}
```

Efficiency Considerations

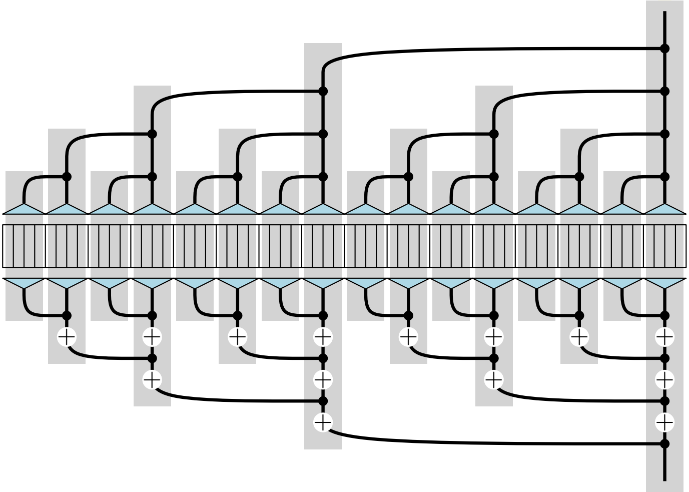
Why use *tasks* instead of *threads*?

Creating and scheduling threads is more expensive.

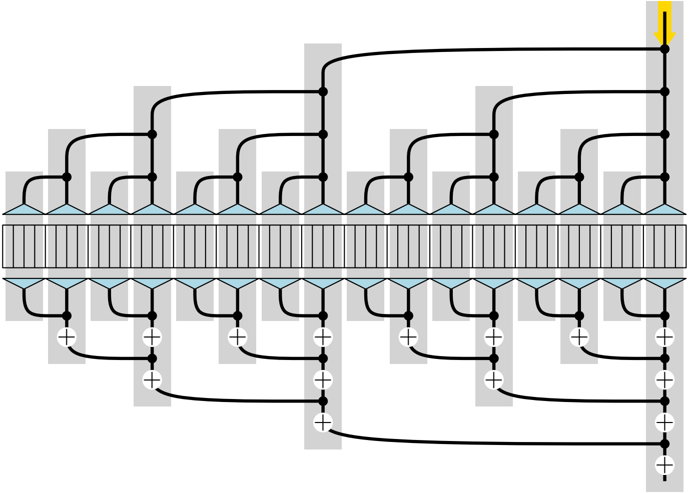
Why use `CUTOFF` to switch to a sequential algorithm?

Creating and scheduling tasks is still somewhat expensive. We want to balance that expense with the amount of work we give the task.

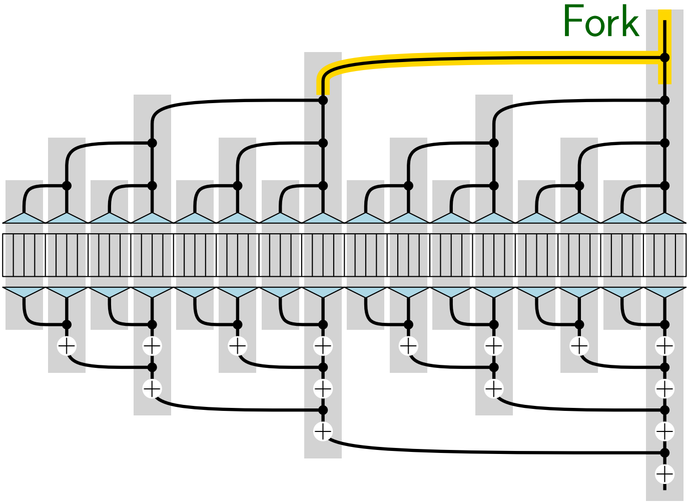
Divide and Conquer in Parallel



Divide and Conquer in Parallel

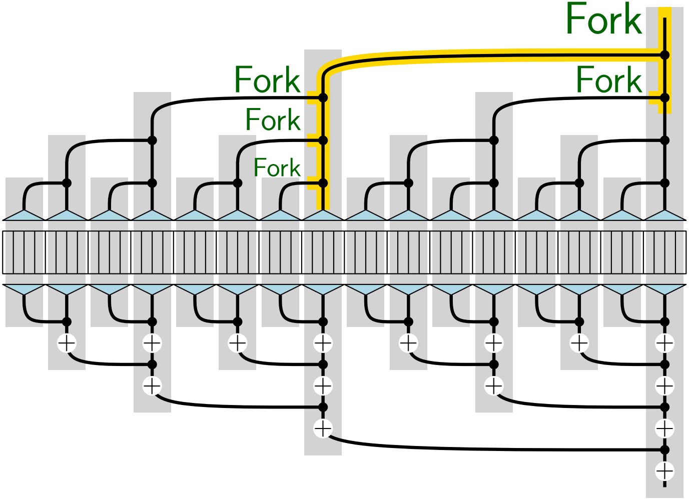


Divide and Conquer in Parallel



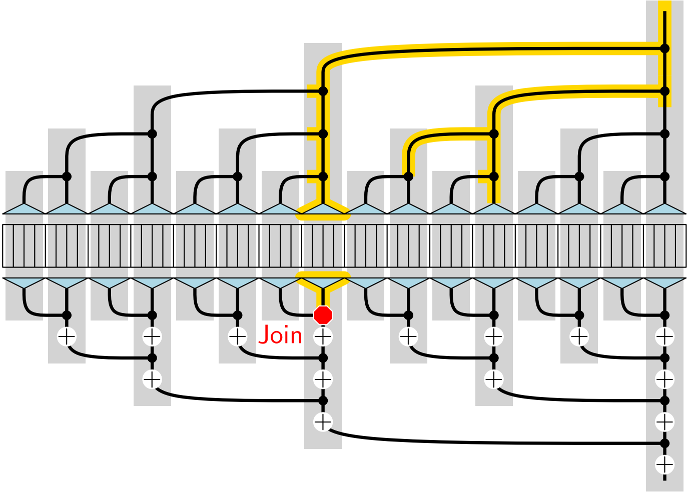
Fork Process creates (forks) a new child process. Both continue executing the same code but they have different IDs.

Divide and Conquer in Parallel



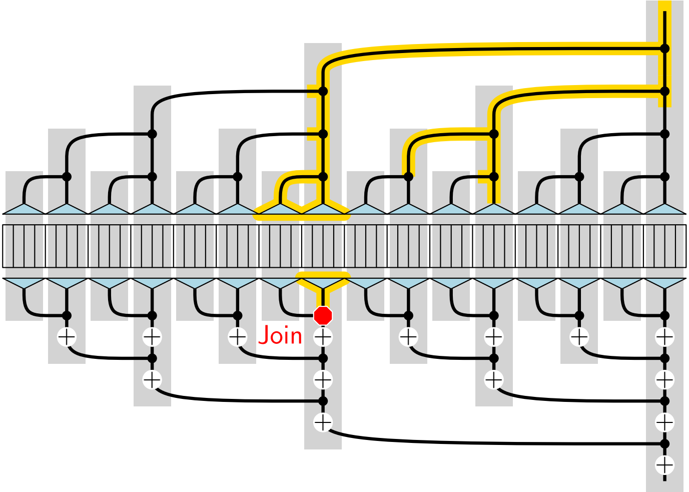
The child and the parent can fork more children.

Divide and Conquer in Parallel



Join Process waits to recombine (join) with its child until the child reaches the same join point.

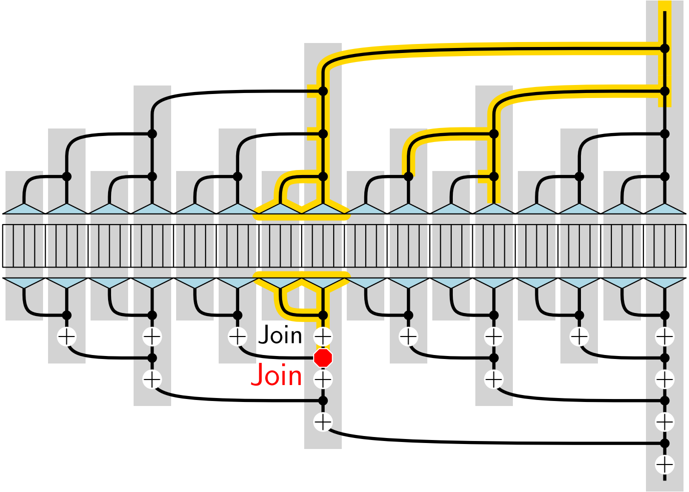
Divide and Conquer in Parallel



Still waiting... Why wait?

Join insures that the child is done before the parent uses its value.

Divide and Conquer in Parallel



After join the child process terminates and the parent continues.

Counting Matches with Divide and Conquer

```
int nmpDC(int A[], int lo, int hi, int key) {
  if( hi - lo <= CUTOFF ) return nMatches(A, lo, hi, key);
  int left, right;
#pragma omp task untied shared(left)   Fork
  { left = nmpDC(A, lo, (lo + hi)/2, key); }
  right = nmpDC(A, (lo + hi)/2, hi, key);
#pragma omp taskwait  Join
  return left + right;
}
int nmpDivConq(int A[], int n, int key) {
  int result;
#pragma omp parallel
#pragma omp single
  { result = nmpDC(A, 0, n, key); }
  return result;
}
```

Efficiency with Many Processors

Let n be the array size and P be the number of processors.

Old Way

1. Divide array into P pieces.
2. Solve these pieces in parallel; time is $\Theta(n/P)$.
3. Combine the pieces by summing the results; time is $\Theta(P)$.

Total time: $\Theta(n/P) + \Theta(P)$

Suppose the number of processors, P, is infinite...

Divide and Conquer Way

1. Recursively divide array into `CUTOFF`-size pieces. Time $\Theta(\log n)$
2. Solve these pieces in parallel; time is $\Theta(\texttt{CUTOFF})$.
3. Combine the pieces by summing the results; time is $\Theta(\log n)$.

Total time: $\Theta(\log n)$

Is Counting Matches Simply a Reduction?

```
FORALL x in A:
    score = (if x == key then 1 else 0)
    total += score
```

FORALL is short for “Do every iteration in parallel”.

```
// OpenMP equivalent
#pragma omp parallel for reduction(+:total)
  for (i=0; i < n; i++)
    total += (A[i] == key) ? 1 : 0;
```

Map

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

- No combining results
- Some hardware supports this directly

Counting matches is a Map (using equalsMap) followed by a Reduce (using $+$).

```
void equalsMap(int score[], int A[], int n, int key) {
  FORALL( i=0; i<n; ++i ) {
    score[i] = (A[i] == key) ? 1 : 0;
  }
}
```

Another Map Example: Vector Addition

$$\langle 1,2,3,4,5\rangle + \langle 2,5,3,3,1\rangle = \langle 3,7,6,7,6\rangle$$

```
void vectorAdd(int sum[], int u[], int v[], int n) {
  FORALL( i=0; i<n; ++i ) {
    sum[i] = u[i] + v[i];
  }
}
```

Parallel Programming by Patterns

Map and Reduce are very common patterns in parallel programs.

Learn to recognize when an algorithm can be written in terms of Map and Reduce. They make parallel programming simple.

By the way... Google's MapReduce and the open-source Hadoop provide parallel Map and Reduce using clusters of computers.

- The system distributes data and manages fault tolerance.
- You provide the Map and Reduce functions.
- This is a case of where the old functional programming ideas (map and fold) take over the world!

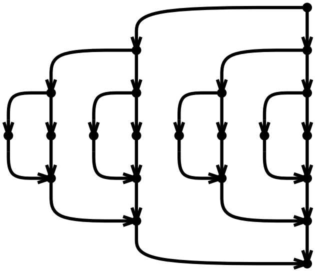
Map / Reduce Exercises

1. Find the ten largest numbers in an array.
2. Count the number of prime numbers in an array of positive integers.
3. Given a (small) pattern string and a (large) text string, find the first occurrence of the pattern in the text.

Modeling Parallel Programs as DAGs

Every parallel program can be modeled as a directed, acyclic graph (DAG).

Nodes represent a constant amount of sequential work.



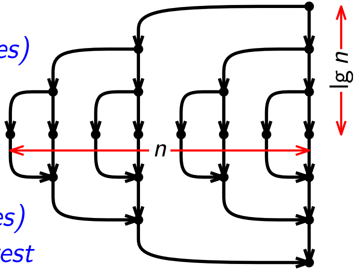
Edges represent dependency: (x, y) means work x must complete before work y starts.

Runtime of Parallel Programs

Let $T_P(n)$ be the running time of a parallel program using P processors on an input of size n.

$T_1(n)$ is called the **Work**

Work equals (some constant times) the number of nodes.



$T_\infty(n)$ is called the **Span**

Span equals (some constant times) the number of nodes on the longest path.

For `nmDivConq` on input of size n (or $n \times$ `CUTOFF`), the number of nodes is $3n - 2$ so $T_1(n) \in \Theta(n)$, and the longest path has $2 \lg n + 1$ nodes so $T_\infty(n) \in \Theta(\log n)$.

Runtime as a Function of n and P

What is $T_P(n)$ in terms of n and P?

- $T_P(n) \geq T_1(n)/P$ because otherwise we didn't do all the work.
- $T_P(n) \geq T_\infty(n)$ because $P < \infty$.

Therefore

$$T_P(n) \in \Omega(\max\{T_1(n)/P, T_\infty(n)\}) = \Omega(T_1(n)/P + T_\infty(n))$$

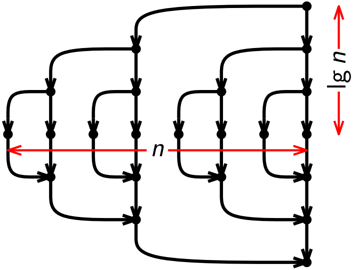
An **asymptotically optimal** runtime is

$$T_p(n) \in \Theta(T_1(n)/P + T_\infty(n)).$$

Good OpenMP implementations guarantee $\Theta(T_1(n)/P + T_\infty(n))$ as their expected runtime.

Runtime of Parallel Divide and Conquer

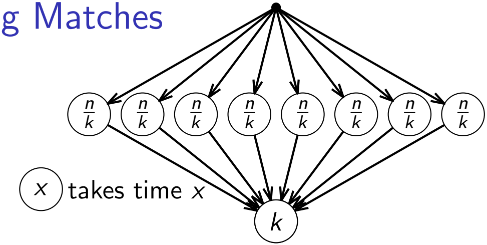
Work $T_1(n) \in \Theta(n)$.
Span $T_\infty(n) \in \Theta(\log n)$.
So $T_P(n) \in \Theta(n/P + \log n)$.



Since Span ($T_\infty(n)$) is so small ($\Theta(\log n)$), our runtime is dominated by n/P for large n.

This means we get linear (in P) speedup over the sequential program, which is the best we could hope for.

Runtime of Parallel Counting Matches



```
omp_set_num_threads(k);
#pragma omp parallel
{
  int id=omp_get_thread_num(), lo = id * nn, hi;
  if (id == k-1) hi = n; else hi = lo + nn;
  results[id] = nMatches(A, lo, hi, key);
}
int result = 0;
for( int i = 0; i < k; i++ ) result += results[i];
return result;
```

Work $T_1(n) \in \Theta(n + k)$ Span $T_\infty(n) \in \Theta(n/k + k)$

Thus, $T_P(n) \in \Theta(\frac{n+k}{P} + n/k + k) \subset \Omega(\sqrt{n})$.

Amdahl's Law

Suppose we know that fraction s of the Work can't be parallelized.

$$T_P(n) \geq sT_1(n) + (1 - s)\,T_1(n)/P$$

since the best we can hope for is linear speedup on the parallel part.

Amdahl's Law The overall speedup with P processors is:

$$\frac{T_1(n)}{T_P(n)} \leq \frac{1}{s + (1 - s)/P}.$$

The overall speedup with ∞ processors is: $\frac{T_1(n)}{T_\infty(n)} \leq \frac{1}{s}$.

Fred Brooks: "Nine women can't make a baby in one month."

Amdahl's Law - Examples

$$\frac{T_1(n)}{T_P(n)} \leq \frac{1}{s + (1-s)/P} \qquad \frac{T_1(n)}{T_\infty(n)} \leq \frac{1}{s}$$

Suppose $s = 33\%$ of a program is sequential.

- What speedup can you get from 2 processors?
- What speedup can you get from 1,000,000 processors?
- Suppose you want 100x speedup with 256 processors?

$$100 \leq \frac{1}{s + (1-s)/256}$$

How small must s be?

Prefix Sum

Given an input array, in, of n numbers, produce an output array, out, where $\text{out}[i] = \text{in}[0] + \text{in}[1] + \cdots + \text{in}[i]$.

For example,

in	42	3	4	7	1	10	5	2
out	42	45	49	56	57	67	72	74

```
vector<int> prefixSum(const vector<int>& in) {
  vector<int> out(in.size());
  out[0] = in[0];
  for( int i=1; i<n; ++i )
    out[i] = out[i-1] + in[i];
  return out;
}
```

Map/Reduce?

Work $T_1(n) \in \Theta(n)$ Span $T_\infty(n) \in \Theta(n)$

Parallel Prefix Sum

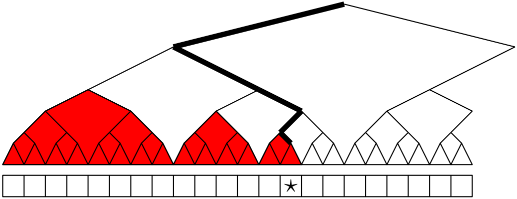
The parallel prefix sum algorithm has two parts:

Part 1 For every subtree of the parallel divide-and-conquer Sum tree, calculate the sums of all leaf entries in the subtree.

> *Easy. That's how the Sum tree works.*

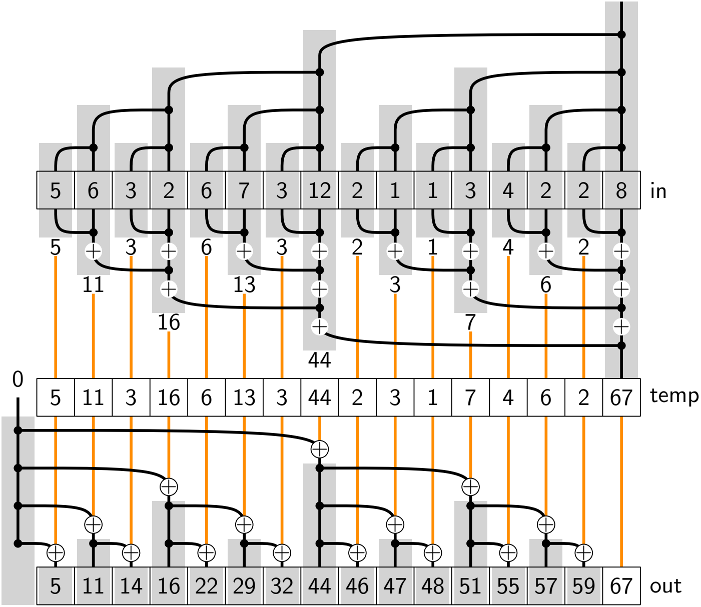
Part 2 Accumulate the sums from disjoint subtrees to the left of each array position.

> *By choosing the biggest subtrees, we accumulate at most* $\lg n + 1$ *subtree sums for each position.*



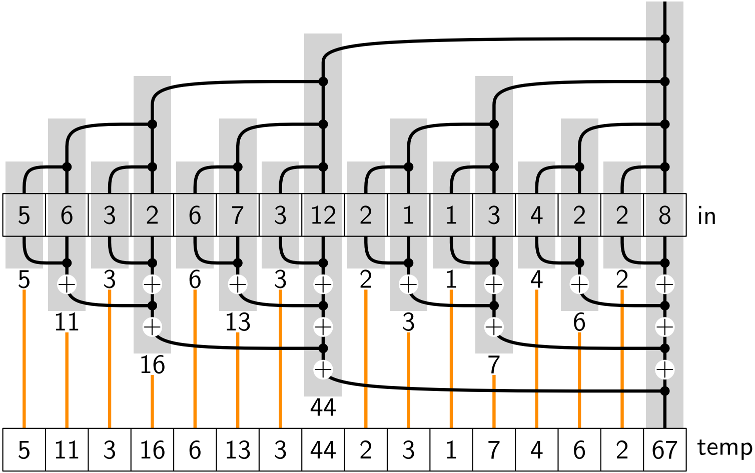
Accumulate all the red subtree sums to get the prefix sum at $\star$.

Parallel Prefix Sum



5	6	3	2	6	7	3	12	2	1	1	3	4	2	2	8	in
5	11	3	16	6	13	3	44	2	3	1	7	4	6	2	67	temp
5	11	14	16	22	29	32	44	46	47	48	51	55	57	59	67	out

Parallel Prefix Sum, Part 1

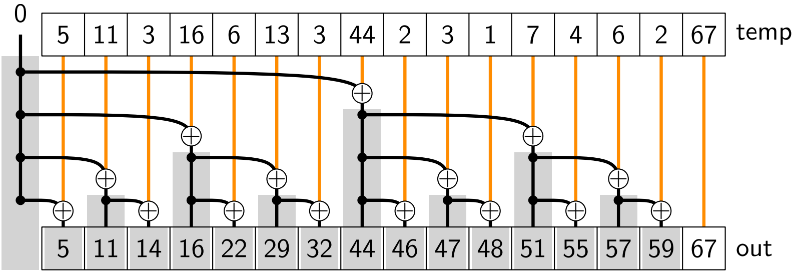


5	6	3	2	6	7	3	12	2	1	1	3	4	2	2	8	in
5	11	3	16	6	13	3	44	2	3	1	7	4	6	2	67	temp

Work $T_1(n) \in \Theta(n)$

Span $T_\infty(n) \in \Theta(\log n)$

Parallel Prefix Sum, Part 2



5	11	3	16	6	13	3	44	2	3	1	7	4	6	2	67	temp
5	11	14	16	22	29	32	44	46	47	48	51	55	57	59	67	out

Work $T_1(n) \in \Theta(n)$

Span $T_\infty(n) \in \Theta(\log n)$

Pack (a.k.a. Filter) using Prefix Sum

Given a predicate (boolean function) p and an array in, produce an array out of those in[i] such that p(in[i]) is true, in the same order that they appear in in.

$$\text{Example:}\quad \begin{array}{l} \text{in} = [17, 4, 6, 8, 11, 5, 13, 19, 0, 24] \\ p(x) = (x > 10) \\ \text{out} = [17, 11, 13, 19, 24] \end{array}$$

1. Map to compute bit-vector of true elements.

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

2. Prefix Sum on the bit-vector.

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

3. Map to produce the output.

```
FORALL( i=0; i<n; ++i )
  if( bits[i] ) out[bitsum[i]-1] = in[i];
```

Work $T_1(n) \in \Theta(n)$ Span $T_\infty(n) \in \Theta(\log n)$