Lecture #4: Asynchronicity, Concurrency, and Parallelism

COSC 3020: Algorithms and Data Structures

Lars Kotthoff[1]
larsko@uwyo.edu

[1] with material from various sources

Unit Outline

- ▷ History and Motivation
- ▷ Asynchronous Execution
- ▷ Parallelism versus Concurrency
- ▷ Counting Matches in Parallel
- ▷ Divide and Conquer
- ▷ Map and Reduce
- ▷ Analyzing Parallel Programs

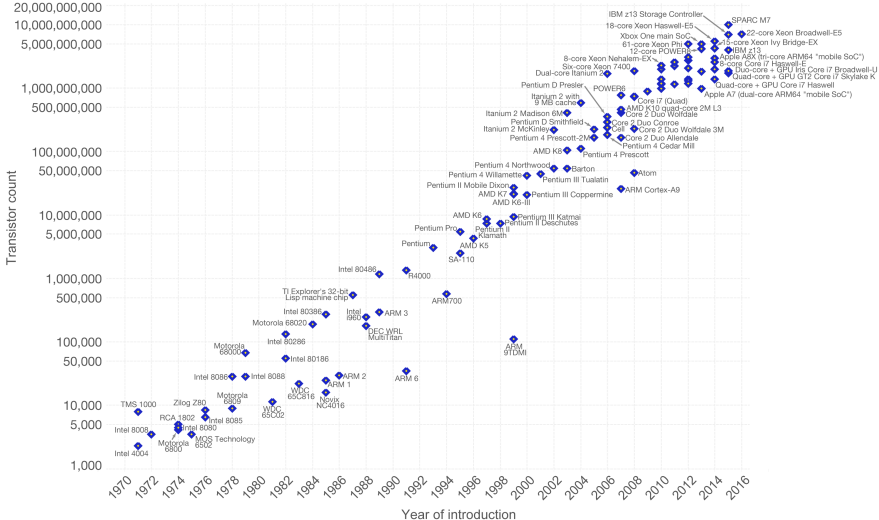
Learning Goals

- ▷ Distinguish between synchronous and asynchronous execution, and know how to write asynchronous programs.
- ▷ 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 and Reduce patterns.

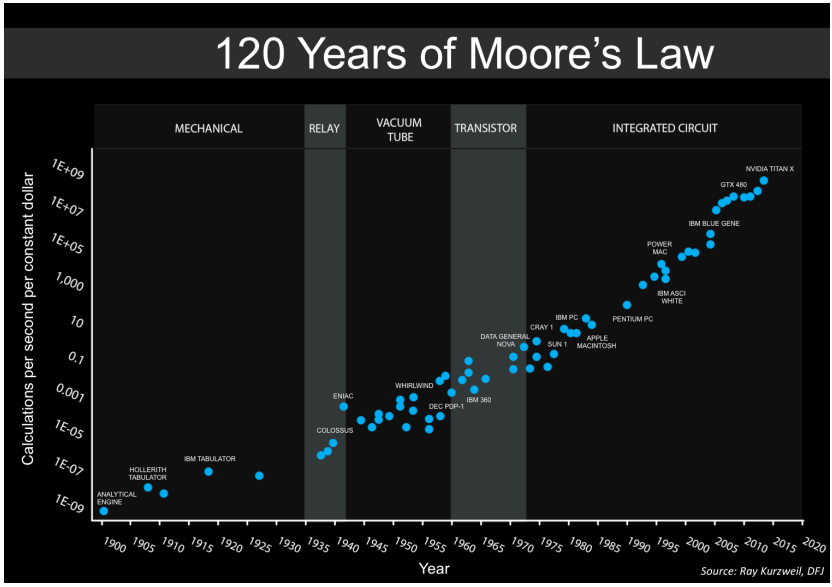
Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

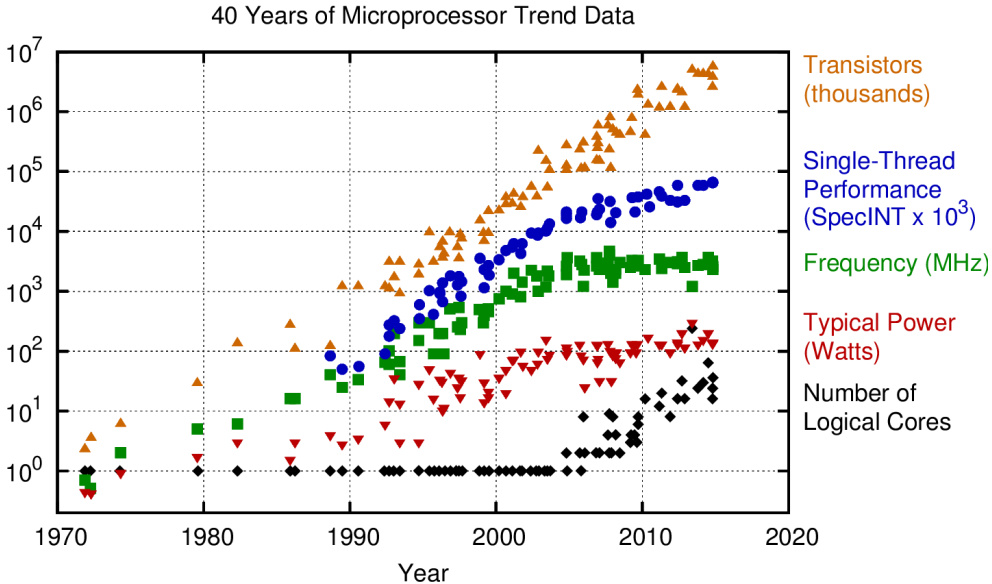
Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.





Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Synchronous Execution

```
function first(){
  console.log(1);
}
function second(){
  console.log(2);
}

first();
second();

// output?
```

Delayed Synchronous Execution

```
function first(){
  setTimeout(function() {
    console.log(1);
  }, 500);
}
function second(){
  console.log(2);
}

first();
second();

// output?
```

Asynchronous Execution – Callbacks

```
function callback() {
  console.log(1);
}

setTimeout(callback, 500);
```

- `callback` that will be run at some point in the future
- function passed as argument to another function
- code is not run in the order in which it is written – asynchronous execution

Asynchronous Execution – Callbacks

General Structure

```
function doStuff(argument, callback) {
  // do something...
  callback();
}
```

Asynchronous Execution – User Interaction

```
$("button").click(function() {
  $("p").hide("slow", function() {
    alert("The paragraph is now hidden.");
  });
});
```

Asynchronous Execution – Network Requests

```
$.get("index.html", function(data) {
  $(".result").html(data);
});
```

Asynchronous Execution

Asynchronous Execution = code will run at some unspecified time

- cannot rely on particular state (e.g. global variables)
- cannot rely on particular order of execution (e.g. callback 1 before 2)
- different callbacks may be executed at the same time (in parallel)
- resources may be accessed concurrently

Parallelism vs. Concurrency

Parallelism

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

Concurrency

Managing access by multiple executing 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.

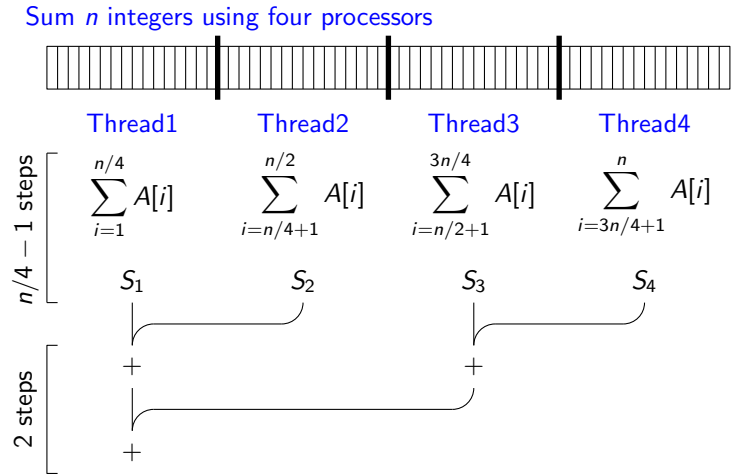
Process Executing instance of a program. The operating system schedules when a process executes on a core.

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

Task Light-weight thread. A task may be scheduled for execution using a different mechanism than the operating system. Not all parallelization frameworks support this.

Parallelism

Performing multiple (computation) steps at the same time.



Total time: $\frac{n}{4}+1$

Concurrency

Managing access by multiple executing agents to a shared resource.

```
var q = [];
function enq(x) {
  let pos = q.length;
  q[pos] = x;
}
```

Concurrency

Managing access by multiple executing agents to a shared resource.

```
var q = [];
function enq(x) {
  let pos = q.length;
  q[pos] = x;
}
```

time	thread 1 enq(1);	thread 2 enq(2);	q
1	pos = q.length;		
2		pos = q.length;	
3		q[pos] = 2;	2
4	q[pos] = 1;		1

Concurrency

Managing access by multiple executing agents to a shared resource.

```
var q = [];
function enq(x) {
  let pos = q.length;
  q[pos] = x;
}
```

time	thread 1 enq(1);	thread 2 enq(2);	q
1	pos = q.length;		
2		pos = q.length;	
3		q[pos] = 2;	2
4	q[pos] = 1;		1

Expected: 1 2

Solution: Mutexes.

Usually, none of this is a problem in Javascript.

Models of Parallel Computation

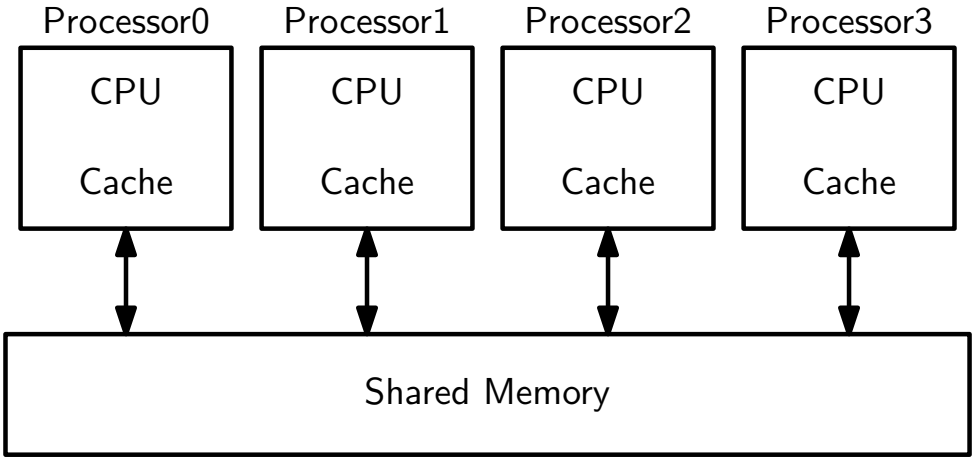
Shared Memory Agents read from and write to a common memory.

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

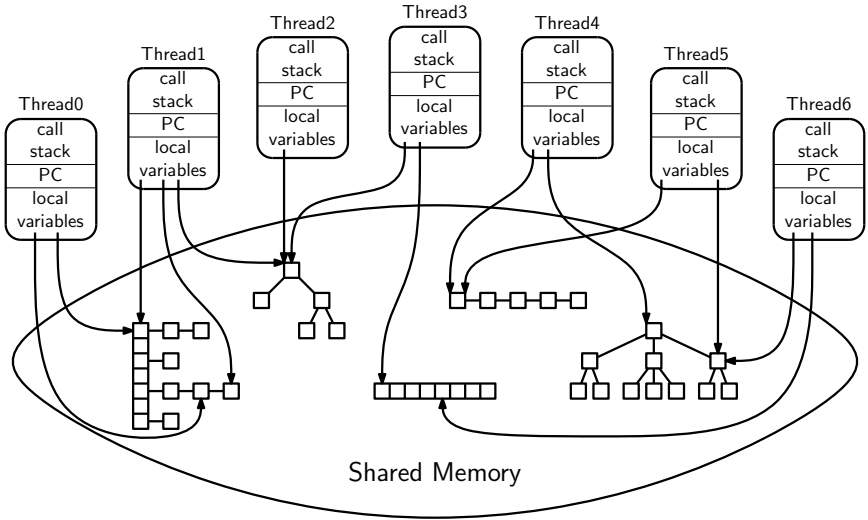
Data Flow Agents are nodes in a directed acyclic graph. Edges represent data that an agent needs as input (incoming) and produces as output (outgoing). When all input is available, the agent can produce output.

Data Parallelism Certain operations (e.g. sum) execute in parallel on collections (e.g. arrays) of data (e.g. Map-Reduce).

Shared Memory in Hardware



Shared Memory in Software



PC = program counter = address of currently executing instruction

Count 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

```
function nMatches(arr, key) {
    let m = 0;
    for(let i = 0;  i < arr.length; i++) {
        if(arr[i] == key) m++;
    }
    return m;
}
```

Count Matches in Parallel – Thread Pool

```
const { StaticPool } = require("node-worker-threads-pool");

function nmPTP(arr, key, done) {
  const threads = 4;
  const pool = new StaticPool({
    size: threads,
    task: function(a) {
        let m = 0;
        for(let i = 0;  i < a.length; i++) {
            if(a[i] == this.workerData) m++;
        }
        return m;
    },
    workerData: key
  });

  const size = arr.length/threads;

  let res = 0, finished = 0;
  for(let i = 0; i < threads; i++) {
    (async () => {
      let r = await pool.exec(arr.slice(i*size, (i+1)*size));
      console.log("Result: " + r);
      res += r;
      finished++;
      if(finished == threads) {
        done(res);
        pool.destroy();
      }
    })();
  }
}
```

How many agents (threads/processes)?

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

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

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

What's the best value of k?

How many agents (threads/processes)?

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

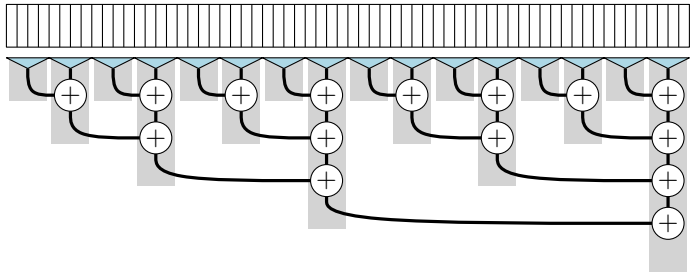
1. Divide array into k pieces of size n/k.
2. Solve these pieces in parallel. Time $\Theta(n/k)$ using k processors
3. Combine by summing the results. Time $\Theta(k)$

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

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

Combine in parallel

The process of producing a single answer[2] from a list is called Reduce.



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

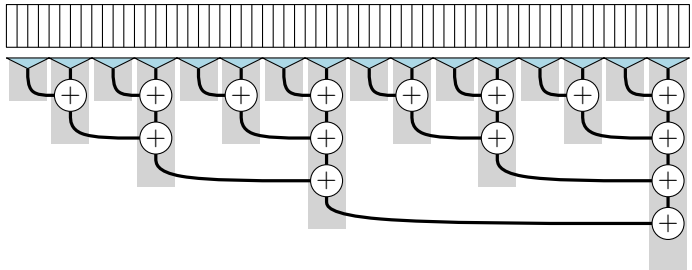
$$a \oplus b \oplus c \oplus d = (a \oplus b) \oplus (c \oplus d)$$

which is true for associative operations.

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

Combine in parallel

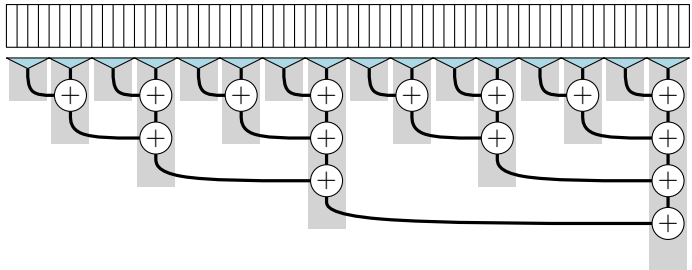
How do we create threads that know how to combine in parallel?



Does this look like anything we've seen before?

Combine in parallel

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!

Count Matches with Divide and Conquer

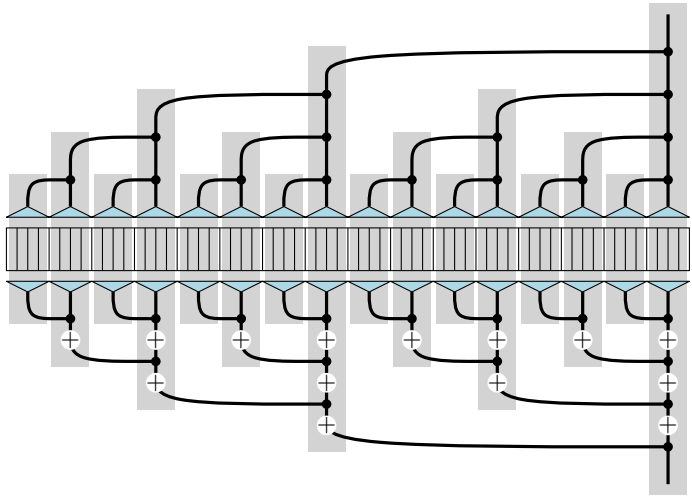
```
function nmPDC(arr, key, done) {
  const fileSync = require('tmp').fileSync;
  const writeFileSync = require('fs').
        writeFileSync;
  const fork = require('child_process').fork;

  function createWorker(fn) {
    const tmpobj = fileSync({ tmpdir: "." });
    writeFileSync(tmpobj.name,
        `process.on('message', function(m) {` +
        `${fn.toString()}` +
        `nmPDC(m[0], m[1]);});`);

    return fork(tmpobj.name);
  }

  function nMatches(arr, key) {
    let m = 0;
    for(let i = 0;  i < arr.length; i++) {
      if(arr[i] == key) m++;
    }
    return m;
  }

  const thresh = 2;
  if(arr.length <= thresh) {
      if(done === undefined) process.send(
           nMatches(arr, key));
      else done(nMatches(arr, key));
      return;
  }

  let left = arr.slice(0, arr.length/2),
      right = arr.slice(arr.length/2, arr.length);

  let res = undefined,
      t = createWorker(nmPDC);

  t.on("message", function(n) {
    console.log("Left worker: " + n);
    if(res === undefined) res = n;
    else {
      if(done === undefined) process.send(res + n);
      else done(res + n);
    }
    t.kill();
  }).send([left, key]);
  nmPDC(right, key, function(n) {
    console.log("Right worker: " + n);
    if(res === undefined) res = n;
    else {
      if(done === undefined) process.send(res + n);
      else done(res + n);
    }
  });
}
```

Efficiency Considerations

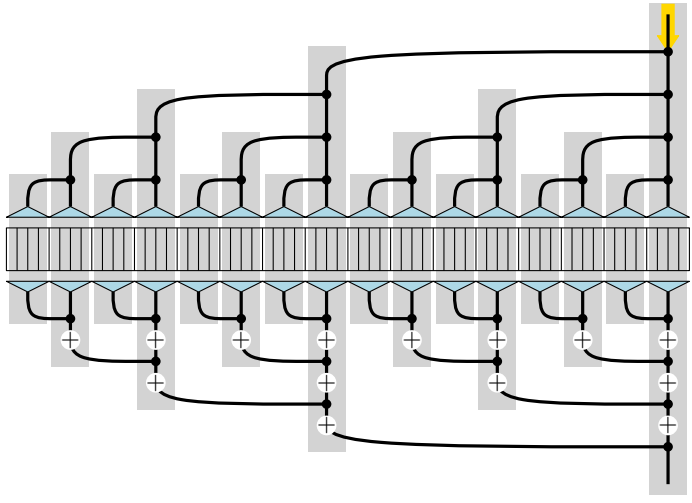
Why use `thresh` to switch to a sequential algorithm?

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

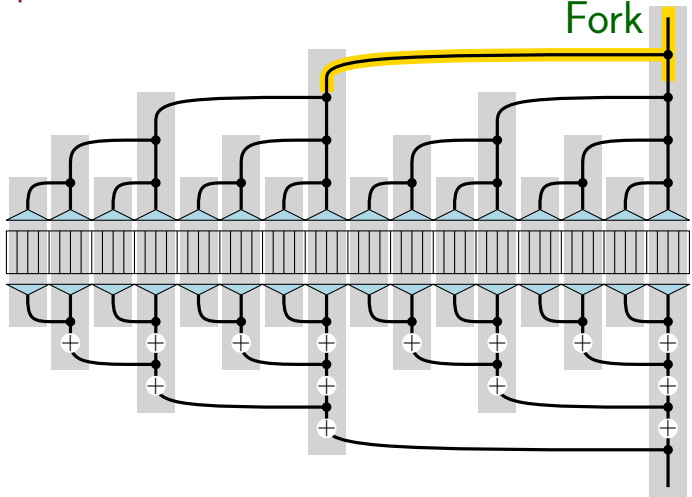
Divide and Conquer in Parallel



Divide and Conquer in Parallel

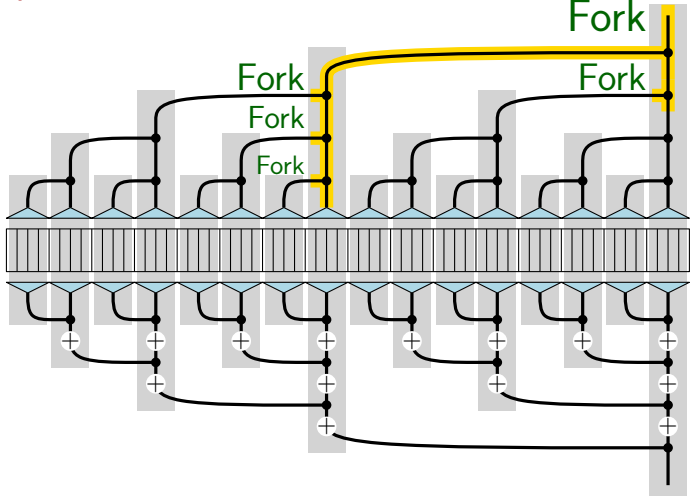


Divide and Conquer in Parallel



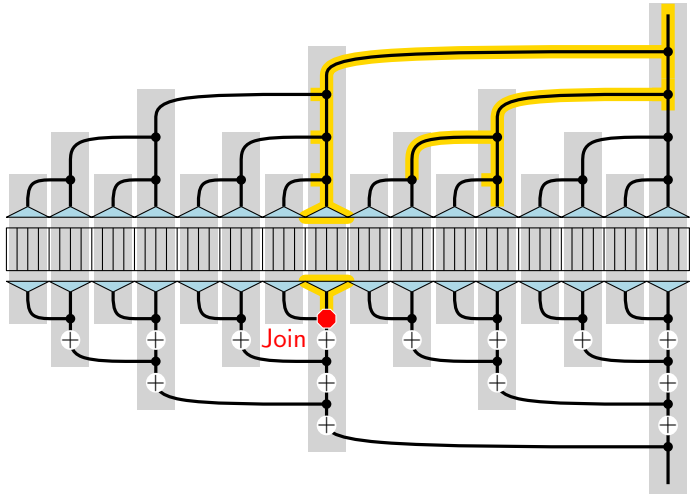
Fork Process creates (spawns) a new child process. Both continue executing the same code but they have different IDs.

Divide and Conquer in Parallel



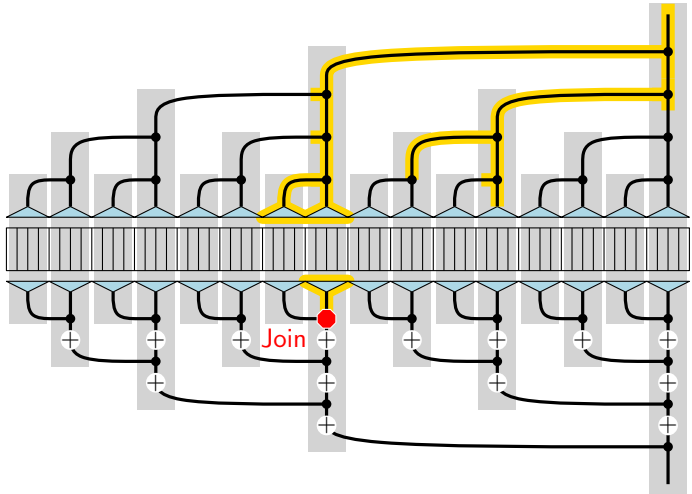
The child and the parent can fork more children.

Divide and Conquer in Parallel



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

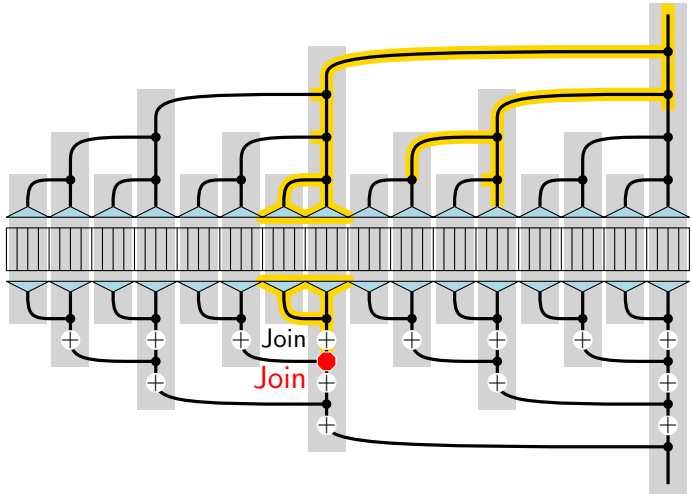
Divide and Conquer in Parallel



Still waiting... Why wait?

Join ensures 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.

Count Matches with Divide and Conquer

```
    let res = undefined,
        t = createWorker(nmPDC);

    t.on("message", function(n) {
      console.log("Left worker: " + n);
      if(res === undefined) res = n;
      else {
        if(done === undefined) process.send(res + n);
        else done(res + n);
      }
      t.kill();
    }).send([left, key]);
    nmPDC(right, key, function(n) {
      console.log("Right worker: " + n);
      if(res === undefined) res = n;
      else {
        if(done === undefined) process.send(res + n);
        else done(res + n);
      }
    });
}
```

Fork

Join

Join

Efficiency with many processors

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

Old Way

1. Divide array into k n/k-size pieces.
2. Solve these pieces in parallel. Time $\Theta(n/k)$
3. Combine by summing the results. Time $\Theta(k)$

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

Efficiency with many processors

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

Old Way

1. Divide array into k n/k-size pieces.
2. Solve these pieces in parallel. Time $\Theta(n/k)$
3. Combine by summing the results. Time $\Theta(k)$

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

Suppose the number of processors is infinite...

Divide and Conquer Way

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

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."

Map

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

- ▷ no combination of results
- ▷ some hardware supports this directly

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

```
function nmPMR(arr, key, cb) {
  var Parallel = require('paralleljs'),
      p = new Parallel(arr);

  global.process.env.key = key;
  p.map(function(i) { return i == global.process.env.key; })
   .reduce(function(d) { return d[0] + d[1]; })
   .then(cb);
}
```

Another Map Example: Vector Addition

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

```
function vaMR(v1, v2, cb) {
  var v = v1.map(function(d, i) {
          return [d, v2[i]];
        }),
      Parallel = require('paralleljs'),
      p = new Parallel(v);

  p.map(function(i) { return i[0] + i[1]; })
   .then(cb);
}
```

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.

- ▷ scale up to large number of processors
- ▷ no concurrency issues
- ▷ correctness of result easily shown

Parallel programming by Patterns

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

- ▷ system distributes data and manages fault tolerance
- ▷ you provide Map and Reduce functions
- ▷ Functional programming ideas (map and fold) take over the world!
- ▷ more in Functional Programming

Map/Reduce Exercises

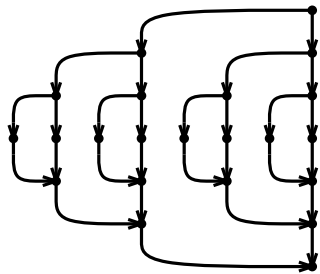
1. Compute the length of each string in an array of strings.
2. Sum all numbers in an array.
3. Count the number of prime numbers in an array of positive integers.

Modeling Parallel Programs as DAGs

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

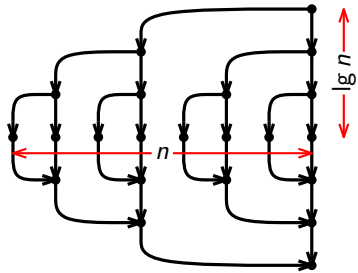
Nodes represent a constant amount of sequential work.

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 `nmPDC` on input of size n (or $n \times$ `thresh`), 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

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

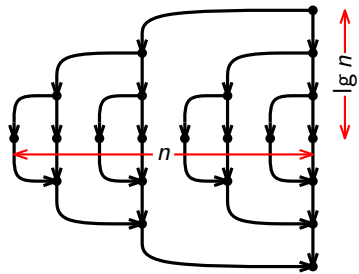
An asymptotically optimal runtime is

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

Good implementations of parallel libraries 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 $T_P(n) \in \Theta(n/P + \log n)$.

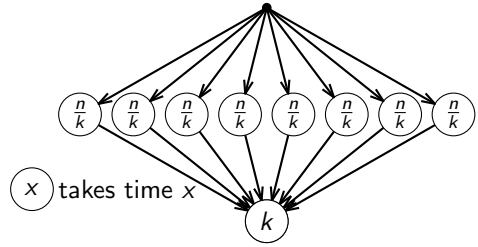


Since Span ($T_\infty(n)$) is so small ($\Theta(\log n)$), our runtime is dominated by n/P for large n.
This means we get linear (in P) speedup over the sequential program, which is the best we could hope for.[3]

[3] Except in special cases.

Runtime of Parallel Count Matches

```
function nmPTP(arr, key, done) {
  const threads = 4;
  const pool = new StaticPool({
    size: threads,
    task: function(a) {
        let m = 0;
        for(let i = 0;  i < a.length; i++) {
            if(a[i] == this.workerData) m++;
        }
        return m;
    },
    workerData: key
  });

  const size = arr.length/threads;

  let res = 0, finished = 0;
  for(let i = 0; i < threads; i++) {
    (async () => {
      let r = await pool.exec(arr.slice(i*size, (i+1)*size));
      console.log("Result: " + r);
      res += r;
      finished++;
      if(finished == threads) {
        done(res);
        pool.destroy();
      }
    })();
  }
}
```



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

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

Amdahl's Law

Suppose we know that s fraction 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: $\frac{T_1(n)}{T_\infty(n)} \leq \frac{1}{s}$

Amdahl's Law

Suppose we know that s fraction 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 $s = \frac{1}{3}$ 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?

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 $s = \frac{1}{3}$ 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? $s < 156/25500 \approx 0.612\%$