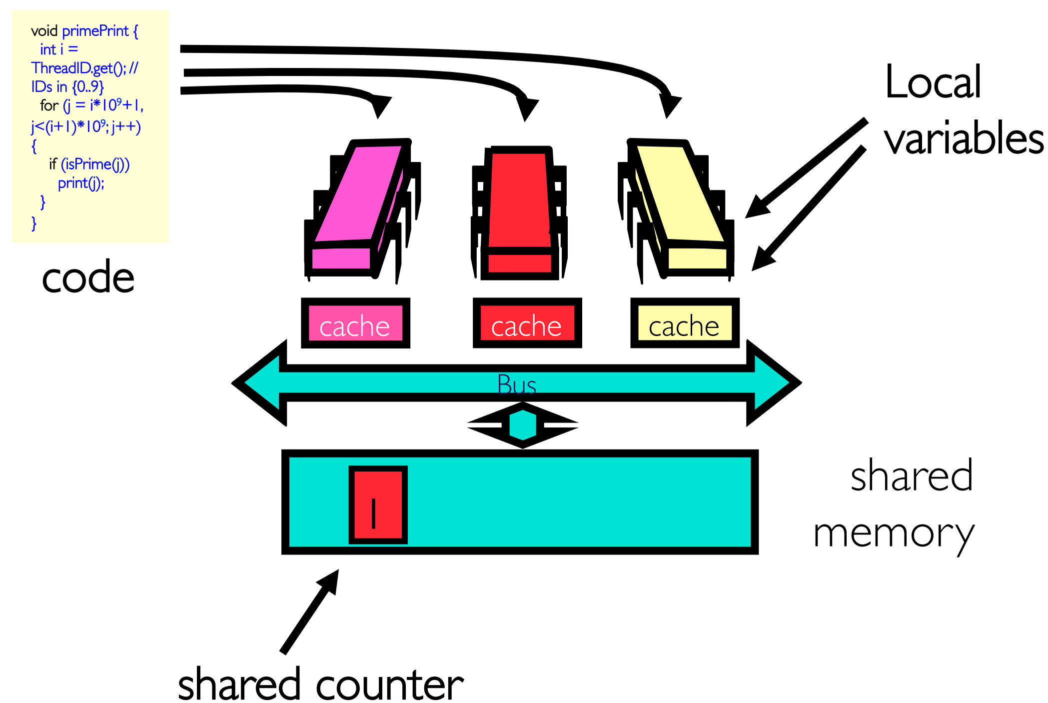
CS-206 Concurrency

Lecture 3
Performance & Efficiency



Spring 2015
Prof. Babak Falsafi
parsa.epfl.ch/courses/cs206/

Adapted from slides originally developed by Maurice Herlihy and Nir Shavit from the Art of Multiprocessor Programming, and Babak Falsafi

Where are We?

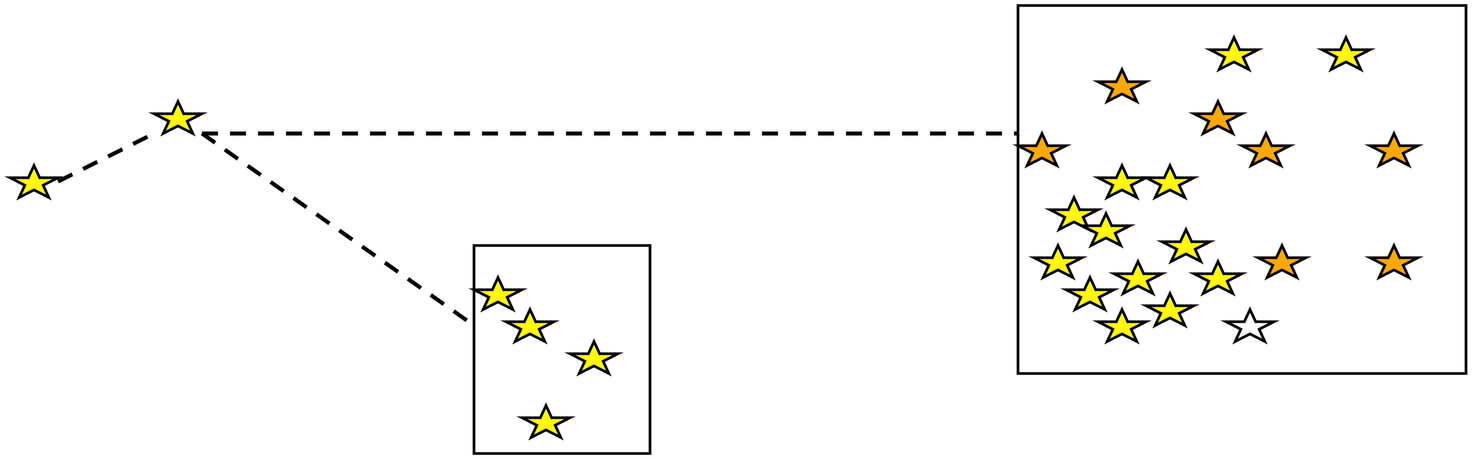
Lecture & Lab

M	T	W	T	F
16-Feb	17-Feb	18-Feb	19-Feb	20-Feb
23-Feb	24-Feb	25-Feb	26-Feb	27-Feb
2-Mar		4-Mar	5-Mar	6-Mar
9-Mar	10-Mar	11-Mar	12-Mar	13-Mar
16-Mar	17-Mar	18-Mar	19-Mar	20-Mar
23-Mar	24-Mar	25-Mar	26-Mar	27-Mar
30-Mar	31-Mar	1-Apr	2-Apr	3-Apr
6-Apr	7-Apr	8-Apr	9-Apr	10-Apr
13-Apr	14-Apr	15-Apr	16-Apr	17-Apr
20-Apr	21-Apr	22-Apr	23-Apr	24-Apr
27-Apr	28-Apr	29-Apr	30-Apr	1-May
4-May	5-May	6-May	7-May	8-May
11-May	12-May	13-May	14-May	15-May
18-May	19-May	20-May	21-May	22-May
25-May	26-May	27-May	28-May	29-May

- Parallelism
  - Parallel thinking
  - Division of work
  - Performance
- Next week
  - Mutual Exclusion

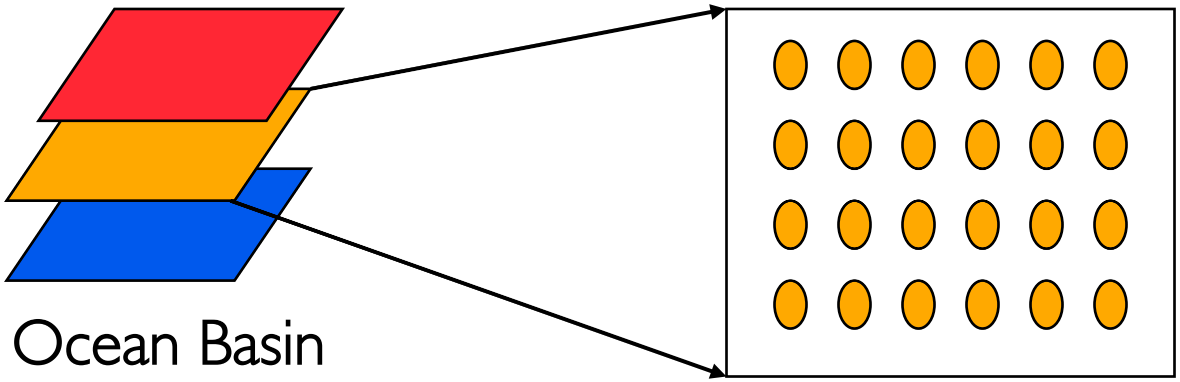
To design well-balanced parallel software we need to think about how a problem can be solved in parallel, divide the work evenly among threads, maximize the parallelism and reduce overhead

Example Problem: Galactic Motion



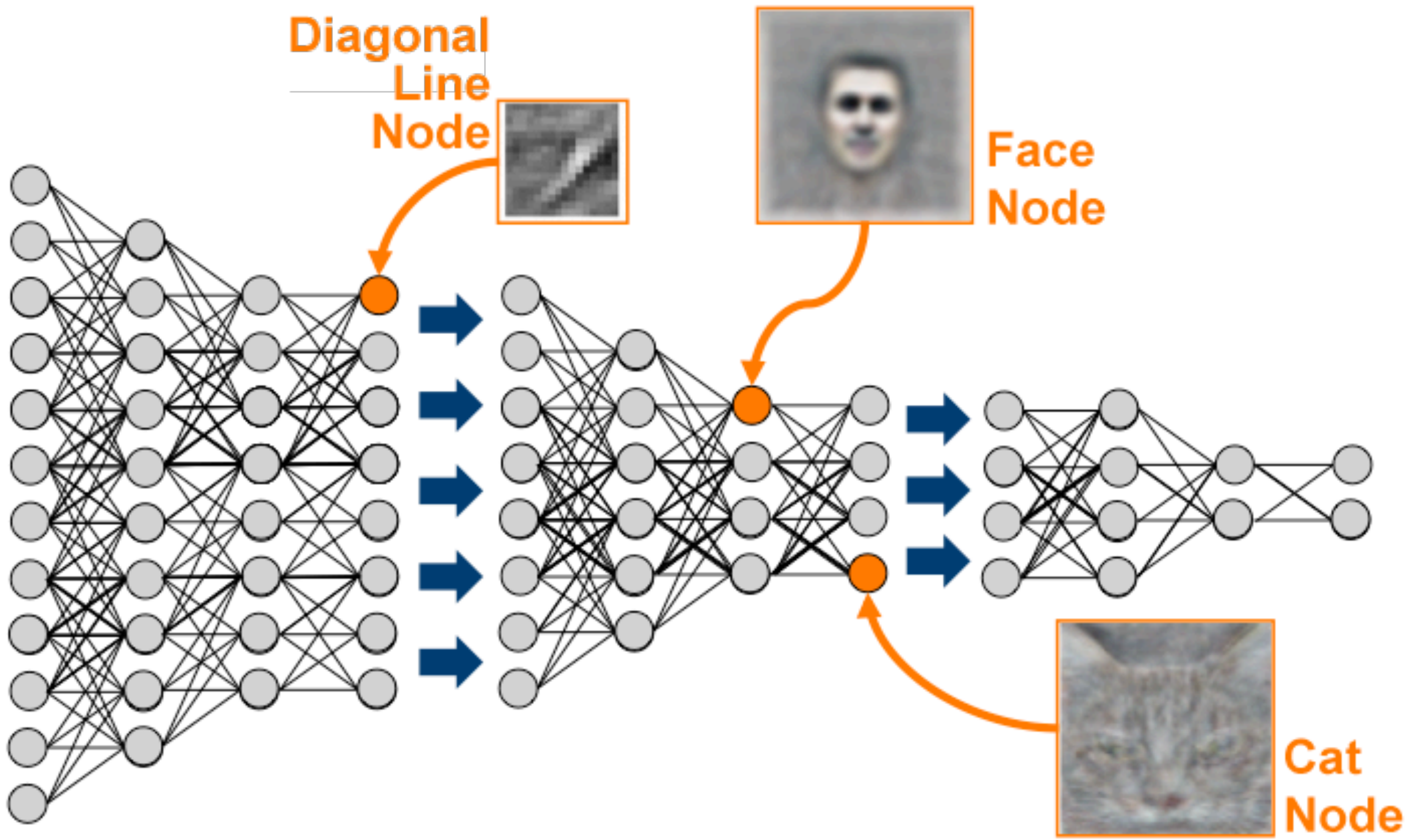
- Computing the mutual interactions of N bodies
  - n-body problems
  - stars, planets, molecules…
  - modeled as a tree
- Can approximate influence of distant bodies
- How do we compute this problem in parallel?

Example Problem: Ocean Simulation



- Simulate ocean eddy currents
- Discretize in space and time
  - Modeled as grids of elements with velocity
  - Compute the impact of near neighbor elements
  - Iterate until a solution is reached
- Used in weather prediction, climate science, …..

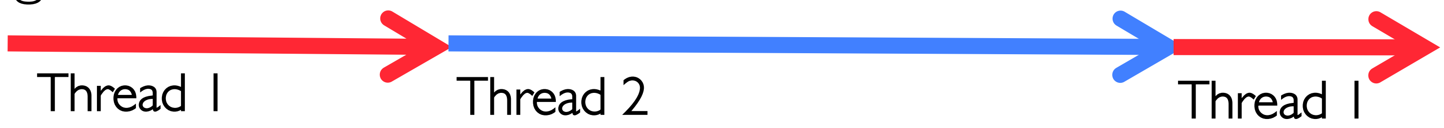
Example Problem: Deep Learning



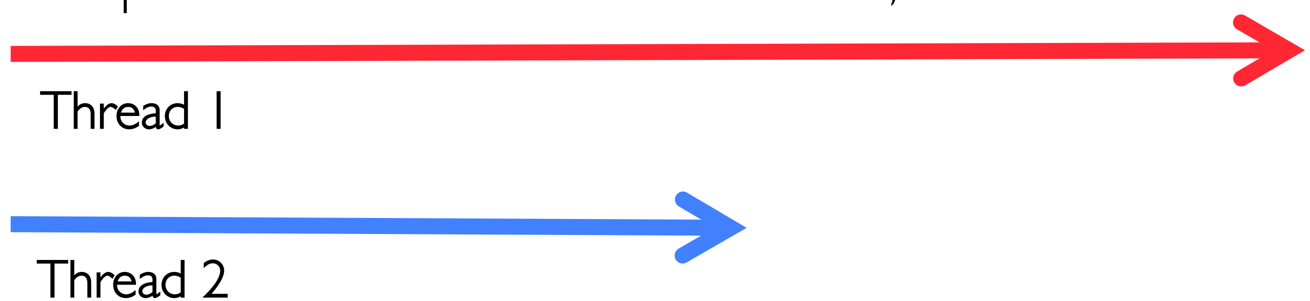
Used in search, machine translation, face recognition, investment banking, …..

Technical distinction: Concurrency vs. Parallelism

- Concurrency: two or more threads make forward progress together



- Parallelism: two or more threads execute at the same time
  - All parallel threads are concurrent, but not vice versa



- Roughly how many threads vs. how many cores

Terminology

- A Task is a piece of work
  - Ocean simulation: compute a grid point, row, plane
  - Raytracing: one ray or group of rays
- Task grain
  - small → fewer operations (less work) per task
  - large → more operations (more work) per task
- Threads performs tasks
  - Threads execute on processors

Forms of Parallelism

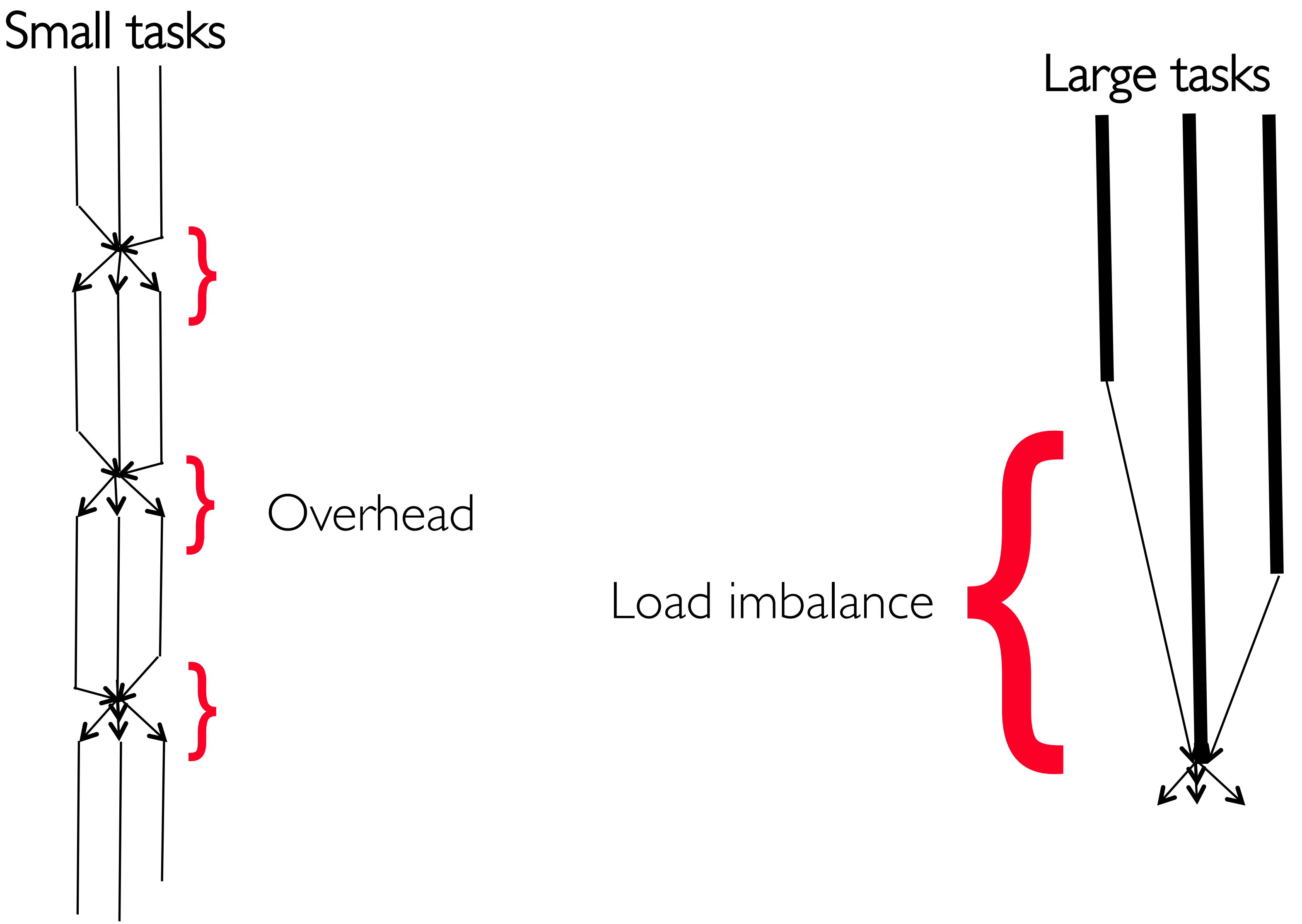
- Throughput parallelism
  - Perform many (identical) sequential tasks at the same time
  - E.g., Google search, ATM (bank) transactions
- Functional or task parallelism
  - Perform tasks that are functionally different in parallel
  - E.g., iPhoto (face recognition with slide show)
- Pipeline parallelism
  - Perform tasks that are different in a particular order
  - E.g., speech (signal, phonemes, words, conversation)
- Data parallelism
  - Perform the same task on different data
  - E.g., Data analytics, image processing

} Reduce time for one job

Division of Work: It's about Performance

- Balance workload
  - Give each parallel task the same rough amount of work
- Reduce communication
  - Balance computation time with communication time
  - Computation → useful work, Communication → overhead
- Reduce extra work
  - Creating a thread, assigning work
  - Scheduling threads on processors, OS, etc.
- These are at odds with each other

Example: Division of Work



Matrix Multiplication

$$\left(\boldsymbol{C}\right) = \left(\boldsymbol{A}\right) \bullet \left(\boldsymbol{B}\right)$$

Matrix Multiplication

$$\left(C_{n\times n}\right)=\left(A_{n\times n}\right)\times\left(B_{n\times n}\right)$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

Where $$c_{ij}=\sum_{k=1}^{n} a_{ik}\cdot b_{kj}$$

Matrix Multiplication

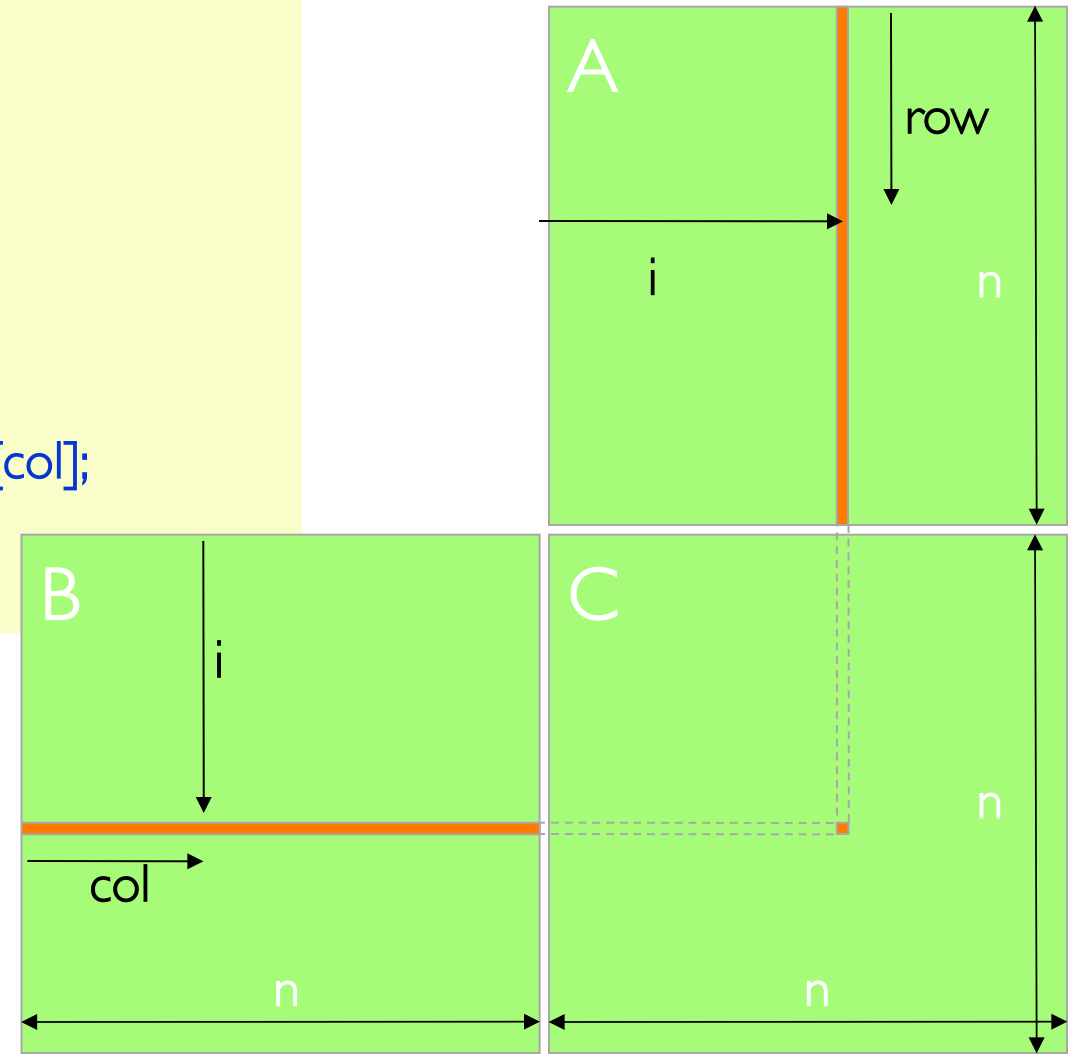
$$\left(C_{n\times n}\right) = \left(A_{n\times n}\right) \times \left(B_{n\times n}\right)$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

For each *i* and *j*:
Multiply the entries a_{ik} by the entries b_{kj} for $k = 1, 2, \ldots, n$ and summing the results over k

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += A[row][i] * B[i][col];
    C[row][col] = dotProduct;
  }}}
```



Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += A[row][i] * B[i][col];
    C[row][col] = dotProduct;
  }}}
```

a thread

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += A[row][i] * B[i][col];
    C[row][col] = dotProduct;
  }}}
```

Which matrix entry to compute

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += A[row][i] * B[i][col];
    C[row][col] = dotProduct;
  }}}
```

Actual computation

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
   for (int row …)
     for (int col …)
       worker[row][col] = new Worker(row,col);
   for (int row …)
     for (int col …)
       worker[row][col].start();
   for (int row …)
     for (int col …)
       worker[row][col].join();
}
```

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
   for (int row …)
     for (int col …)
       worker[row][col] = new Worker(row,col);
   for (int row …)
     for (int col …)
       worker[row][col].start();
   for (int row …)
     for (int col …)
       worker[row][col].join();
}
```

Create n x n threads

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row …)
    for (int col …)
      worker[row][col] = new Worker(row,col);
  for (int row …)
    for (int col …)
      worker[row][col].start();
  for (int row …)
    for (int col …)
      worker[row][col].join();
}
```

Start them

Matrix Multiplication

```
void multiply() {
 Worker[][] worker = new Worker[n][n];
  for (int row …)
   for (int col …)
    worker[row][col] = new Worker(row,col);
  for (int row …)
   for (int col …)
    worker[row][col].start();
  for (int row …)
   for (int col …)
    worker[row][col].join();
}
```

Start them

Wait for them to finish

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row …)
    for (int col …)
      worker[row][col] = new Worker(row,col);
  for (int row …)
    for (int col …)
      worker[row][col].start();
  for (int row …)
    for (int col …)
      worker[row][col].join();
}
```

Start them

Wait for them to finish

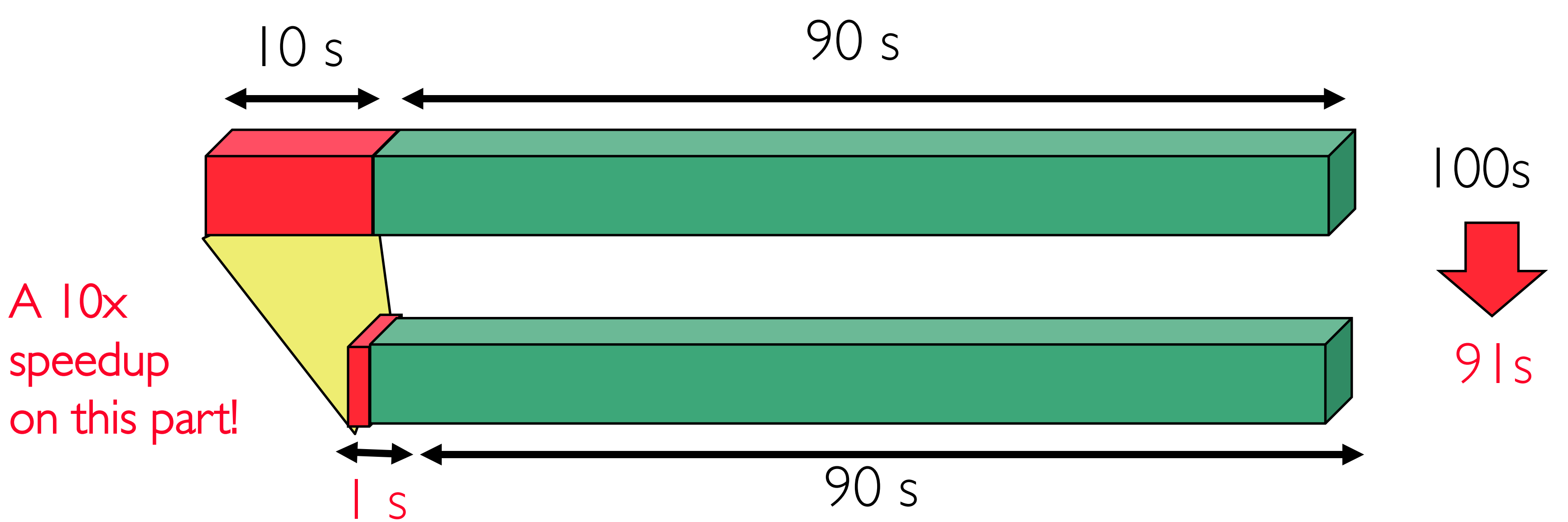
What's wrong with this picture?

Thread Overhead

- One thread per task
  - One dot product task
- Threads Require resources
  - State:
    - Memory for stacks
    - A copy of the register file
    - Program state: Program Counter, Stack pointer,….
  - Setup, teardown
  - Scheduler overhead
- Short-lived threads
  - Bad ratio of work versus overhead

One More "Big" Performance-Related Axiom

- Amdahl's Law
  - In English: if you speed up only a small fraction of the execution time of a program or a computation, the speedup you achieve on the whole application is limited
- Example



Amdahl's Law

Parallel fraction

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

Amdahl's Law

Sequential fraction

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

Amdahl's Law

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

Simple example:

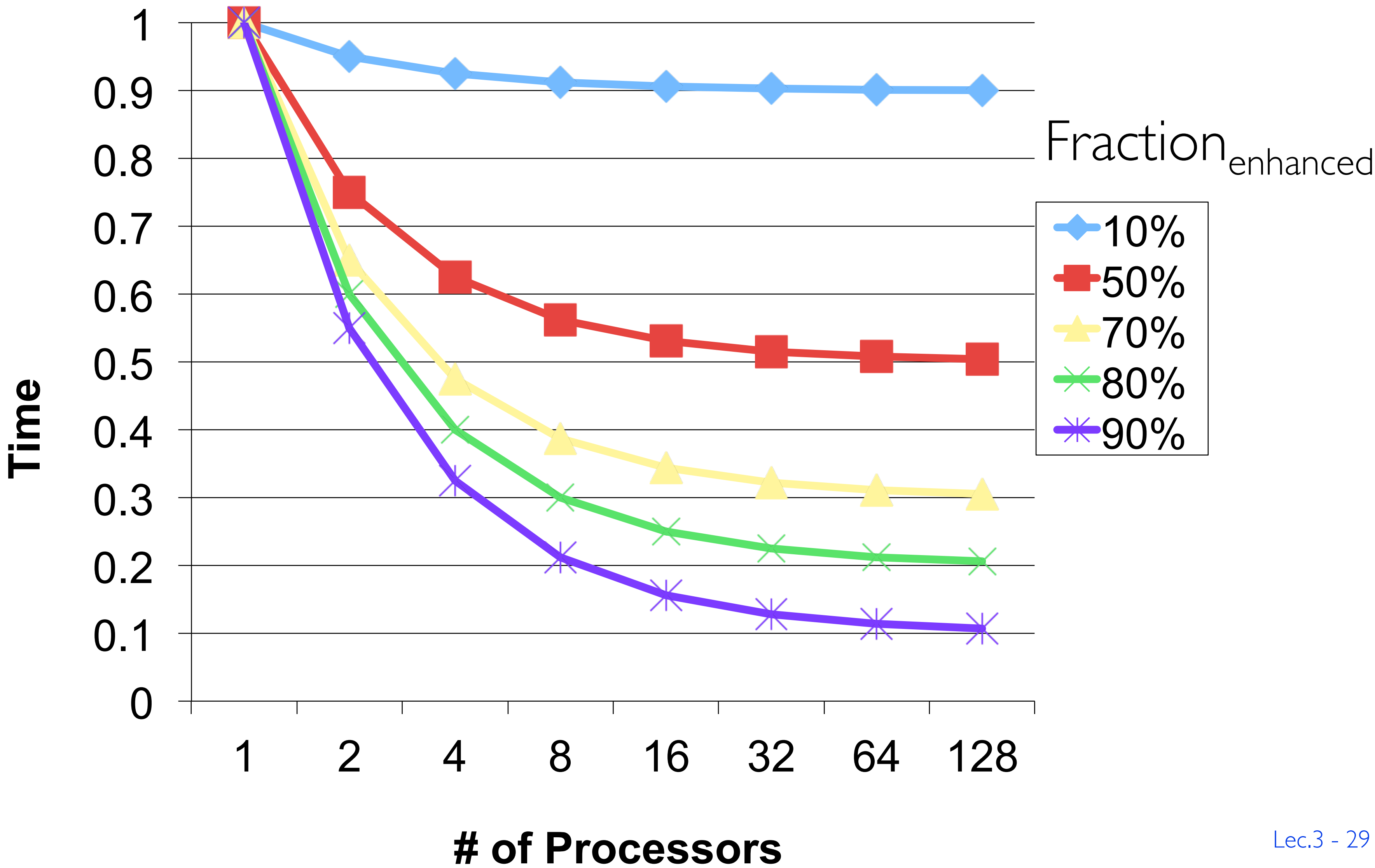
Program runs for 100 seconds on a uniprocessor

10% of the program can be parallelized on a multiprocessor

Assume an ideal multiprocessor with 10 processors

$$\text{Speedup} = \frac{1}{\frac{0.1}{10} + (1-0.1)} = \frac{1}{0.01 + 0.9} = \frac{1}{0.91} = 1.1$$

Implications of Amdahl's Law



Parallel execution is not ideal

- 10 processors rarely get a speedup of 10
  - Load imbalance
  - Thread start/join overhead
  - Communication overhead
- Even if $\text{Fraction}_{\text{enhanced}}$ is close to 100%
  - $\text{Speedup}_{\text{enhanced}}$ << p for p processors
  - Our goal is to get it as close as possible to p

Amdahl's Law (in practice)

Back to Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row …)
    for (int col …)
      worker[row][col] = new Worker(row,col);
  for (int row …)
    for (int col …)
      worker[row][col].start();
  for (int row …)
    for (int col …)
      worker[row][col].join();
}
```

Sequential

$(1 - \text{Fraction}_{\text{enhanced}})$

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += A[row][i] * B[i][col];
    C[row][col] = dotProduct;
  }}}
```

Parallel ($Fraction_{enhanced}$)

Example: $n = 16$

- How many threads will our Matrix Multiplication create?

- How many of these threads are concurrent (i.e., degree of concurrency)?

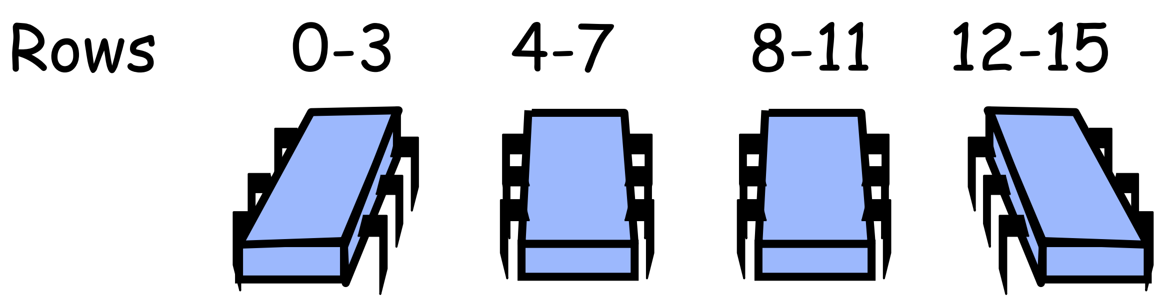
Example: Assume there are 4 processors

- How many threads per processor?

- How many threads run in parallel (i.e., degree of parallelism)?

Best efficiency: Concurrency ~ Parallelism

- All work is independent
- Max parallelism? 4
- Only need 4 threads
  - Reduce thread.start(), thread.join() overhead
  - Only 4 start() and 4 join()
  - Workers (each thread) should do more work
  - 16x16 dot products divided by 4 = 64 dot products per thread



Matrix Multiplication on "p" cores

```
void multiply() {
  BigWorker[] worker = new BigWorker[n];

  for (int row=0; row < n; row+=n/p)
        worker[row] = new BigWorker(row);

  for (int row=0; row < n; row+=n/p)
         worker[row].start();

  for (int row=0; row < n; row+=n/p)
         worker[row].join();
}
```

Matrix Multiplication: (n/c) x n per worker

```
void multiply() {
  BigWorker[] worker = new BigWorker[n];

  for (int row=0; row < n; row+=n/p)
       worker[row] = new Worker(row);

  for (int row=0; row < n; row+=n/p)
       worker[row].start();

  for (int row=0; row < n; row+=n/p)
       worker[row].join();
}
```

Create p threads

BigWorker: Each thread does (n/p) x n

```
class BigWorker extends Thread {
  int begin_row;
  BigWorker(int begin_row) {
    this.begin_row = begin_row;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int row=begin_row; row < begin_row+n/p; row++)
          for (int col=0; col < n; col++)
              for (int i = 0; i < n; i++)
                  dotProduct += A[row][i] * B[i][col];
              C[row][col] = dotProduct;
  }}}
```

BigWorker: Each thread does (n/p) x n

```
class Worker extends Thread {
  int begin_row;
  BigWorker(int begin_row) {
    this.begin_row = begin_row;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int row=begin_row; row < begin_row+n/p; row++)
      for (int col=0; col < n; col++)
        for (int i = 0; i < n; i++)
          dotProduct += A[row][i] * B[i][col];
          C[row][col] = dotProduct;
}}}
```

Multiple rows
All columns

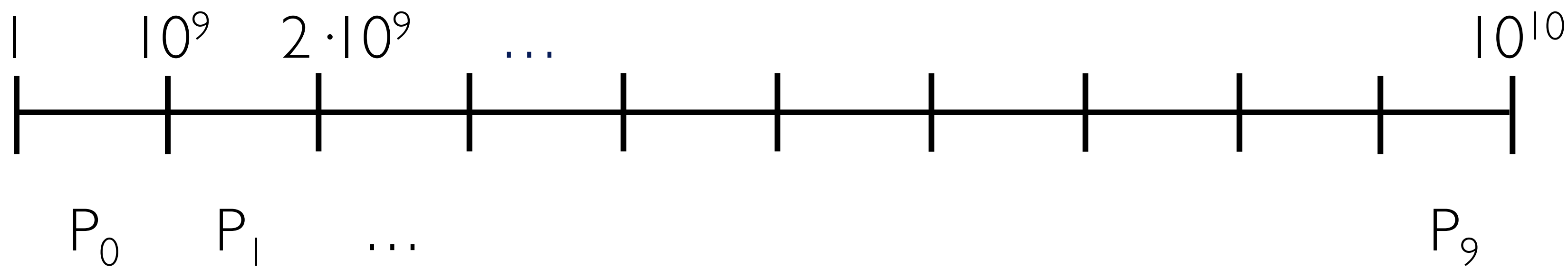
What if n is not divisible by p?

- Lets pick n = 20 and p = 16
  - Matrices of 20x20 running on 16 cores

Parallel Primality Testing

- Challenge
  - Print primes from 1 to 10^{10}
- Given
  - Ten-processor multiprocessor
  - One thread per processor
- Goal
  - Get ten-fold speedup (or close)

Load Balancing



- Split the work evenly
- Each thread tests range of 10^9

Procedure for Thread *i*

```
void primePrint {
  int i = ThreadID.get(); // IDs in {0..9}
  for (j = i*10^9+1, j<(i+1)*10^9; j++) {
    if (isPrime(j))
      print(j);
  }
}
```

Issues

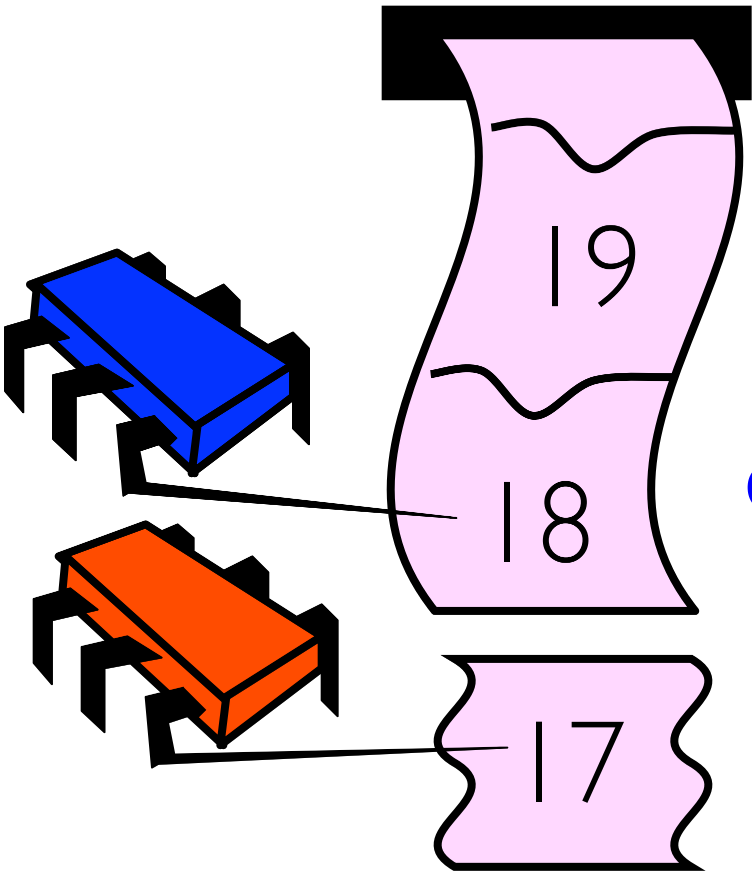
- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict

Issues

- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict
- Need *dynamic* load balancing

rejected

Shared Counter



each thread takes a number

Procedure for Thread *i*

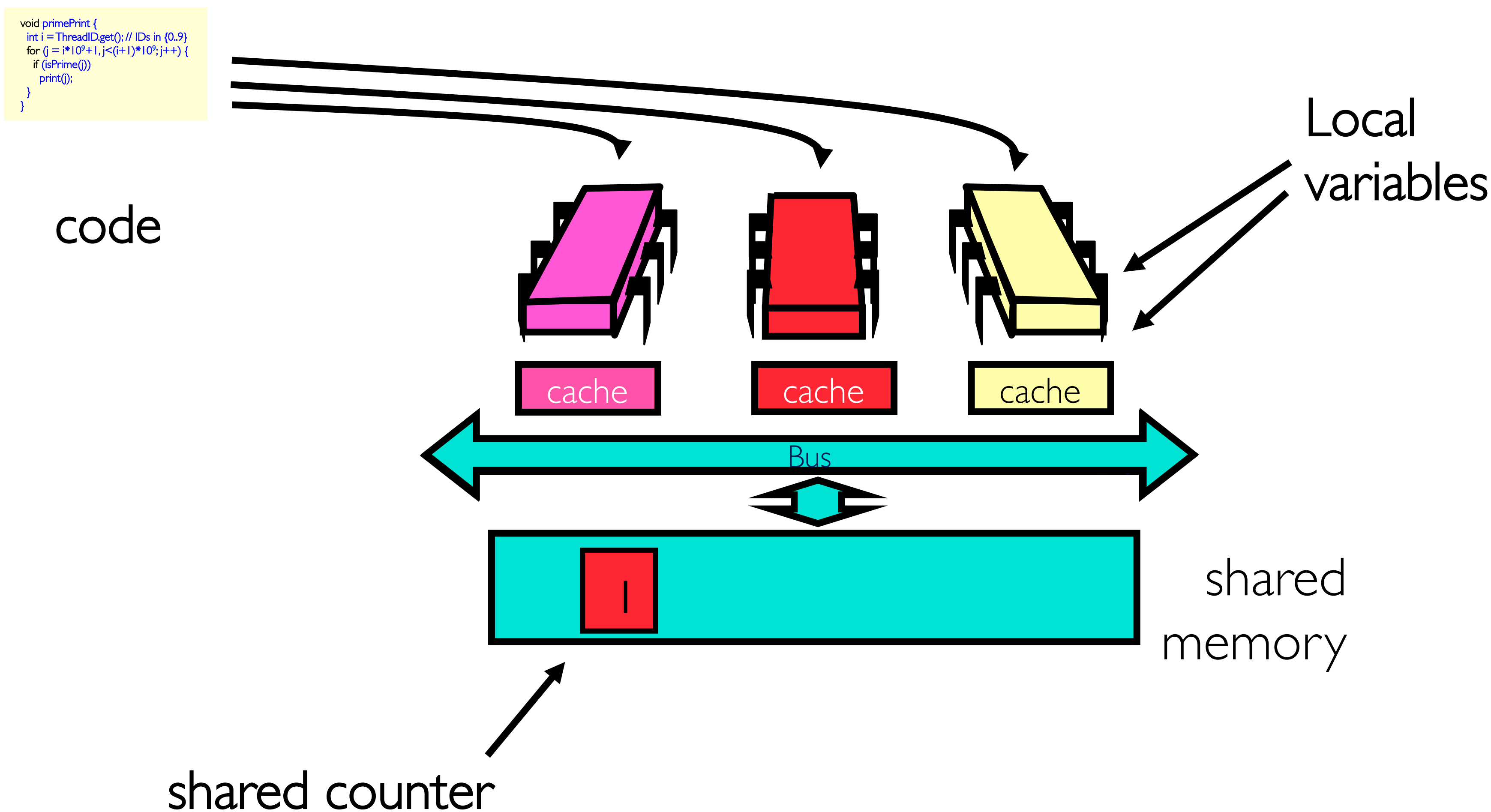
```
int counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Procedure for Thread *i*

```
Counter counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Shared counter object

Where Things Reside



Procedure for Thread *i*

```
Counter counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Stop when every value taken

Procedure for Thread *i*

```
Counter counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Increment & return each new value

Counter Implementation

```java
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

OK for single thread,
not for concurrent threads

What It Means

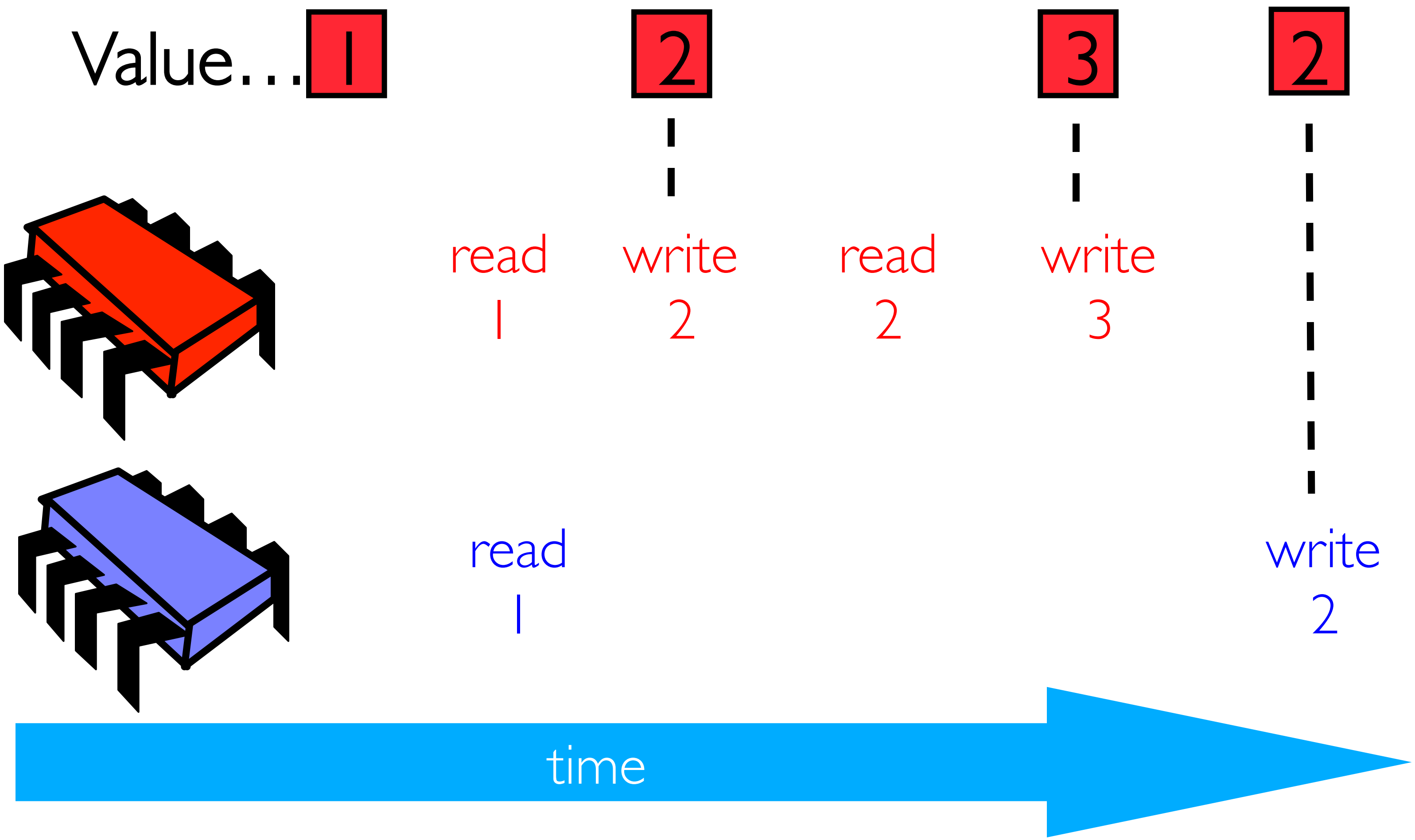
```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

What It Means

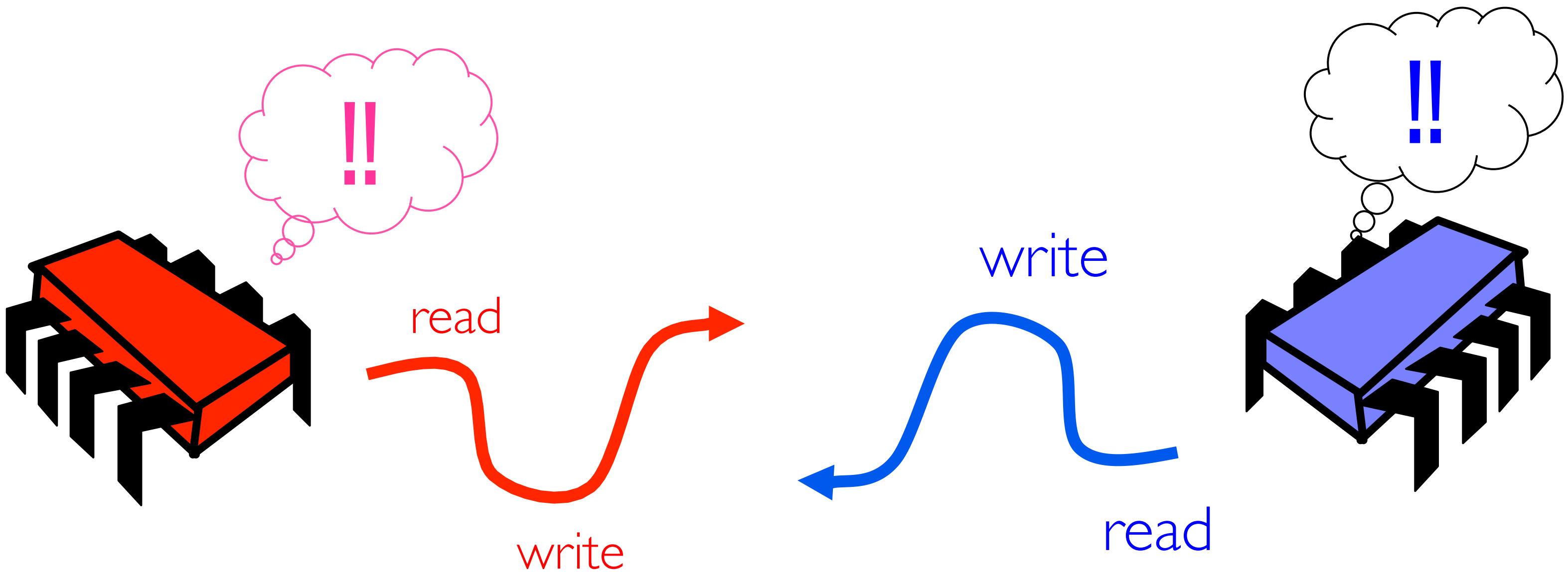
```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

```
temp  = value;
value = temp + 1;
return temp;
```

Not so good…



Is this problem inherent?



If we could only glue reads and writes together…

Challenge

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```

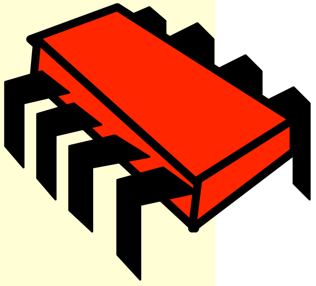
Challenge

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```

Make these steps *atomic* (indivisible)

Hardware Solution

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```



ReadModifyWrite()
instruction

An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
      }
    return temp;
  }
}
```

An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
      }
    return temp;
  }
}
```

Synchronized block

An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
    }
    return temp;
  }
}
```

Mutual Exclusion

Summary

- Need to think parallel
  - Division of work
  - Lots of bottlenecks
- Don't forget Amdahl's Law
- Next week, Mutual Exclusion