



CSCI-UA.0480-003  
**Parallel Computing**

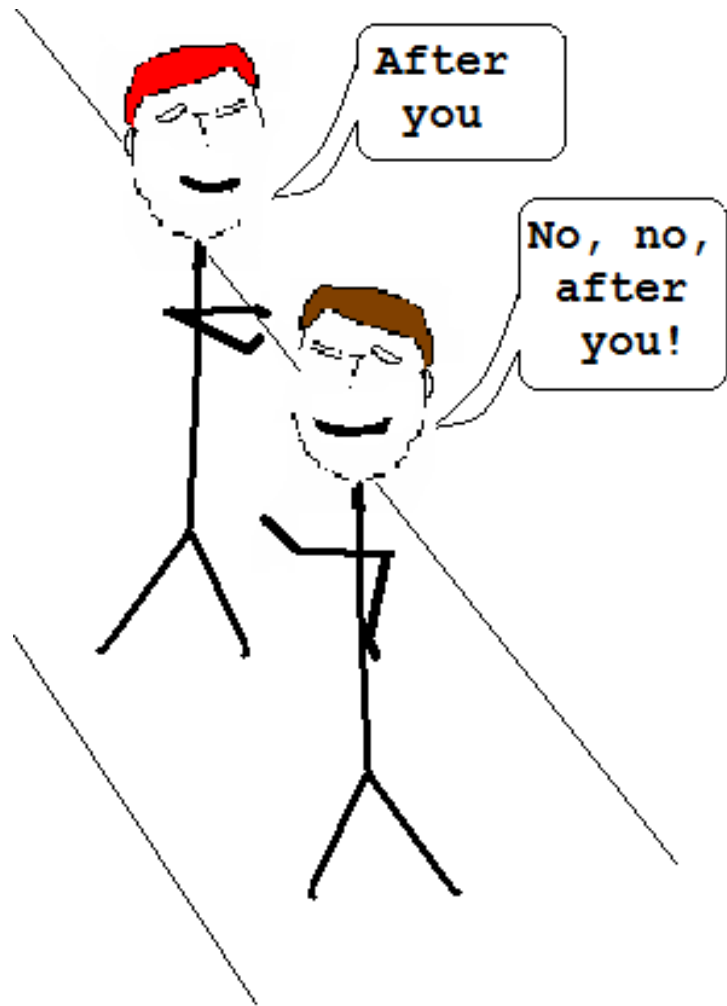
# Lecture 5: Parallel Software: Advanced

Mohamed Zahran (aka Z)  
mzahran@cs.nyu.edu  
<http://www.mzahran.com>

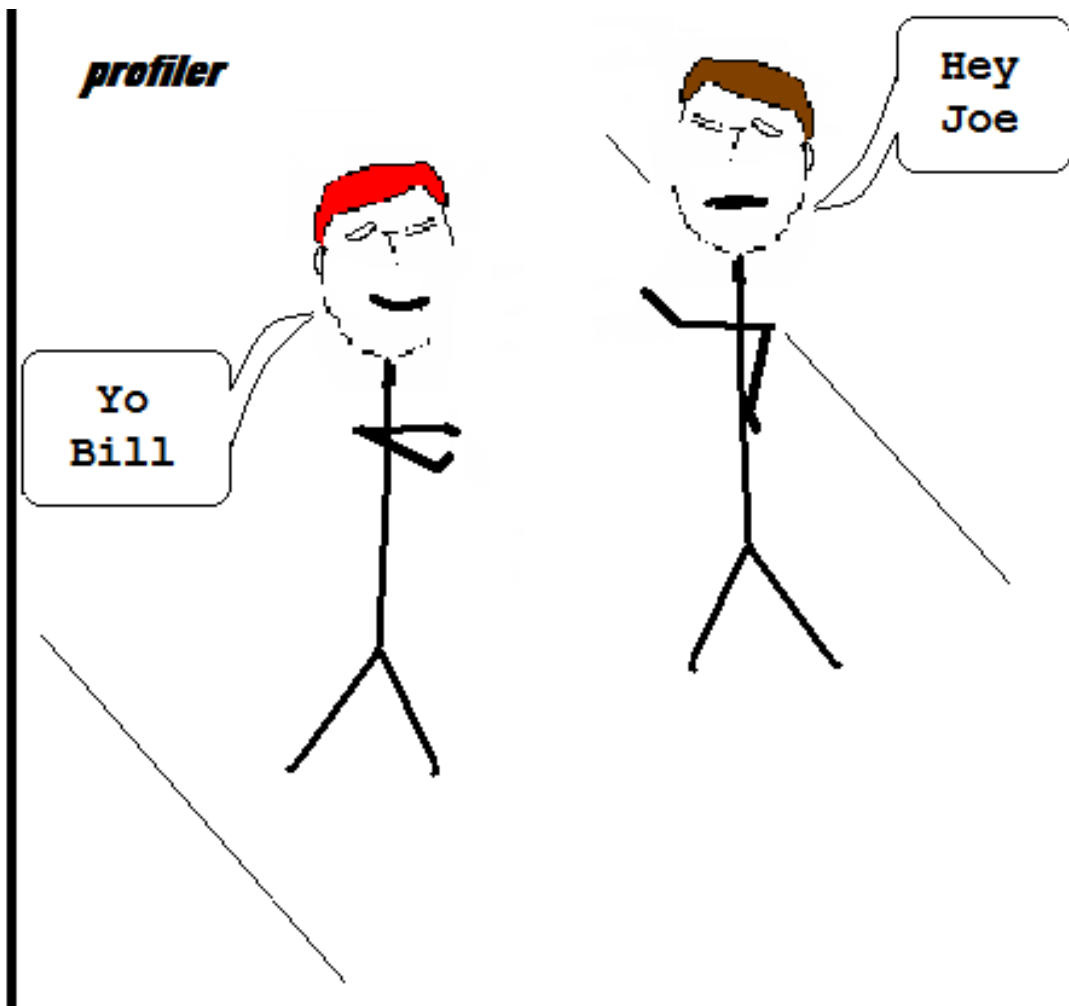


# Concurrency Vs Parallelism: Same Meaning?

- **Concurrency**: At least two tasks are making progress at the same time frame.
  - Not necessarily at the same time
  - Include techniques like time-slicing
  - Can be implemented on a single processing unit
  - Concept more general than parallelism
- **Parallelism**: At least two tasks execute *literally* at the same time.
  - Requires hardware with multiple processing units



Concurrency without parallelism



Concurrency with parallelism

Performance tuning technique number 106: Concurrency vs. Parallelism

Simply Speaking

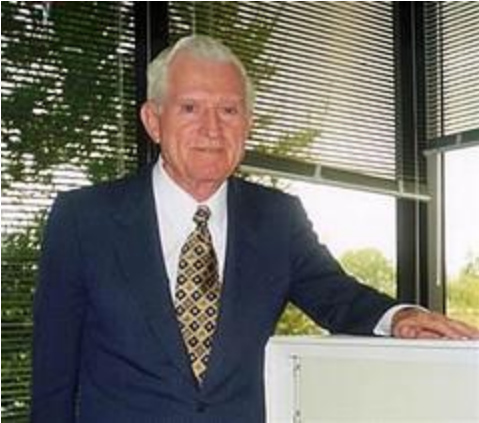
**Concurrency + Parallelism**  
**=**  
**Performance**

# Questions!

If we have as much hardware as we want,  
do we get as much parallelism as we wish?

If we have 2 cores, do we get 2x speedup?

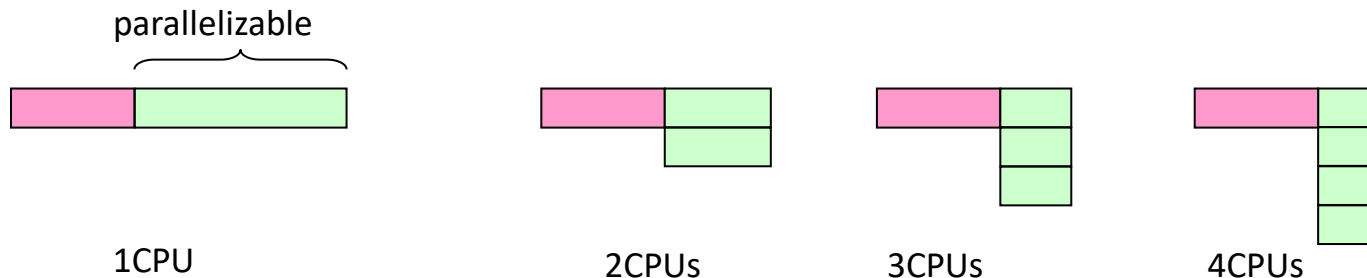
# Amdahl's Law



Gene M. Amdahl

- How much of a speedup one could get for a given parallelized task?

If  $F$  is the fraction of a calculation that is sequential then the maximum speed-up that can be achieved by using  $P$  processors is  $1/(F+(1-F)/P)$



# What Was Amdahl Trying to Say?

- Don't invest blindly on large number of processors.
- Having faster core (or processor at his time) makes more sense than having many cores.

## Was he right?

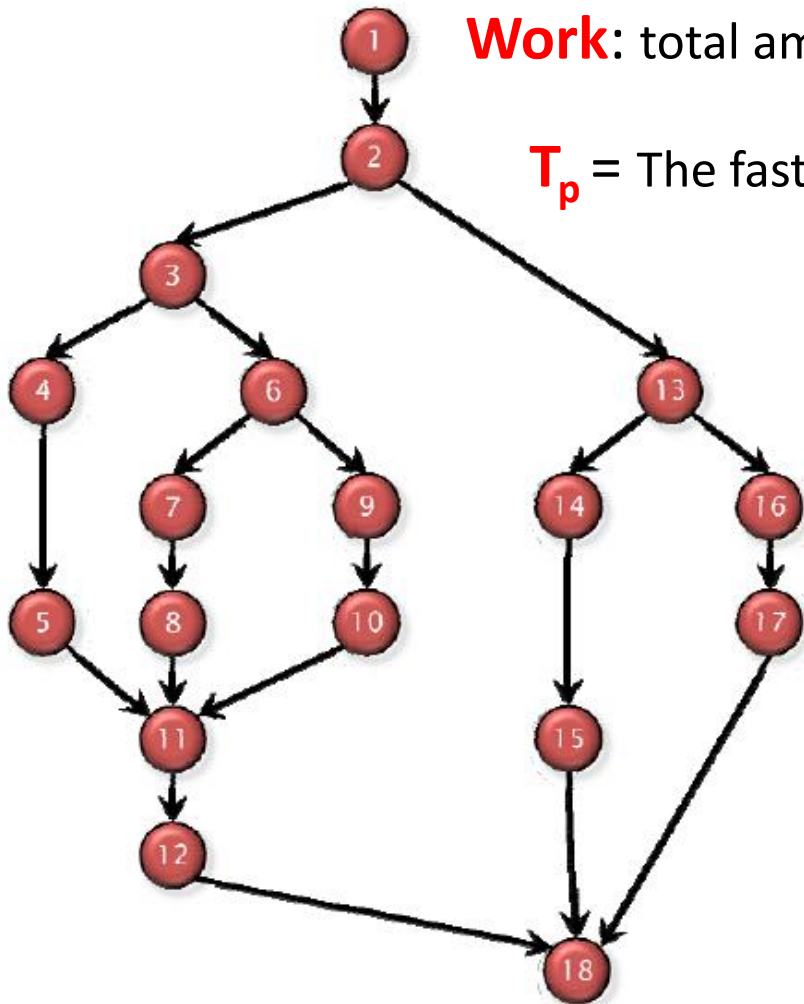
- At his days (the law appeared 1967) many programs have long sequential parts.
- This is not necessarily the case nowadays.
- It is not very easy to find  $F$  (sequential portion)

# So ...

- Decreasing the serialized portion is of greater importance than adding more cores
- Only when a program is mostly parallelized, does adding more processors help more than parallelizing the remaining rest
- Amdahl does not take into account:
  - The overhead of synchronization, communication, OS, etc.
  - Load may not be balanced among cores
- So you have to use this law as guideline and theoretical bound only.



# DAG Model for Multithreading



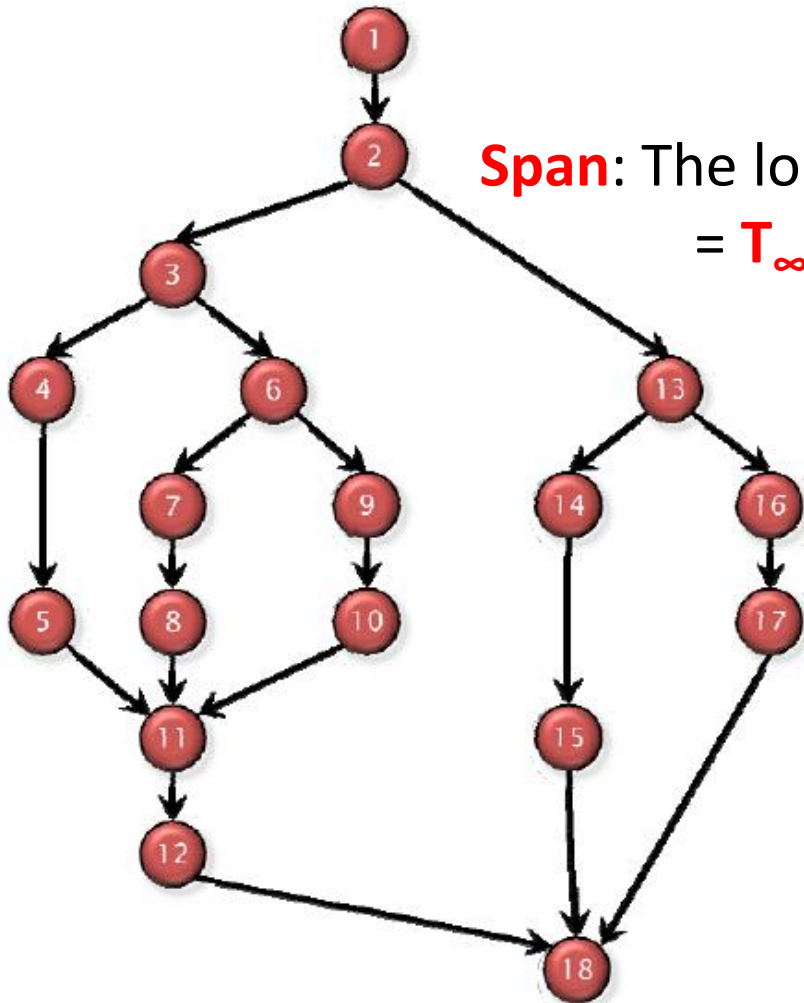
**Work:** total amount of time spent on all instructions

$T_p$  = The fastest possible execution time on P processors

*Work Law:*

$$T_p \geq T_1/P$$

# DAG Model for Multithreading



**Span:** The longest path of dependence in the DAG  
 $= T_{\infty}$

*Span Law:*

$$T_p \geq T_{\infty}$$

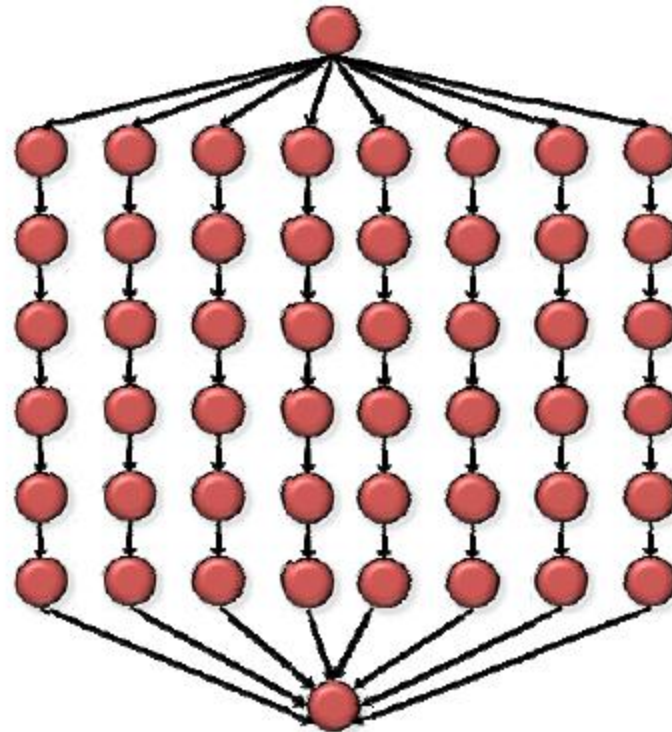
# Can We Define Parallelism Now?

How about?

$$T_1/T_\infty$$

Ratio of work to span

# Can We Define Parallelism Now?



Assume every node is an instruction that takes 1 cycle.

**Work:**  $T_1 = 50$

**Span:**  $T_\infty = 8$

**Parallelism:**  $T_1/T_\infty = 6.25$

# Programming Model

- **Definition:** the languages and libraries that create an abstract view of the machine
- Control
  - How is parallelism created?
  - How are **dependencies** enforced?
- Data
  - Shared or private?
  - How is shared data accessed or private data communicated?
- Synchronization
  - What operations can be used to coordinate parallelism
  - What are the atomic (indivisible) operations?

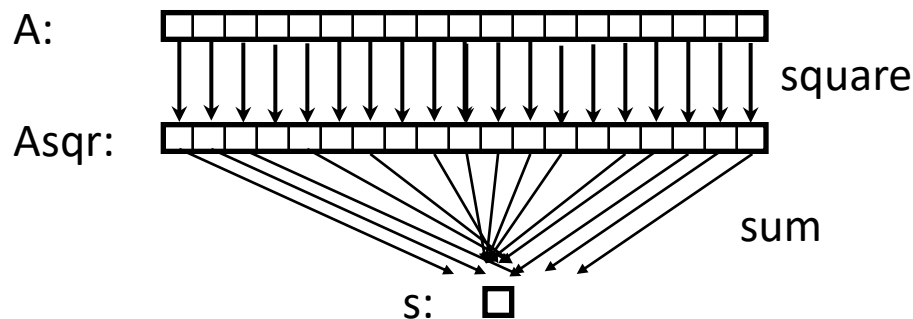
# It Is Important to Note

- You can run any paradigm on any hardware
- The hardware itself can be heterogeneous

The whole challenge of parallel programming is to make the best use of the underlying hardware to exploit the different type of parallelisms

# Example

We have a matrix  $A$ . We need to form another matrix  $Asqr$  that contains the square of each element of  $A$ . Then we need to calculate  $S$ , which is the sum of the elements in  $Asqr$ .



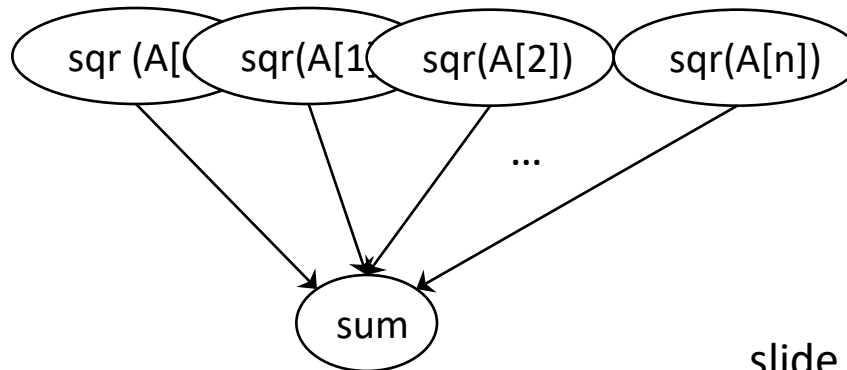
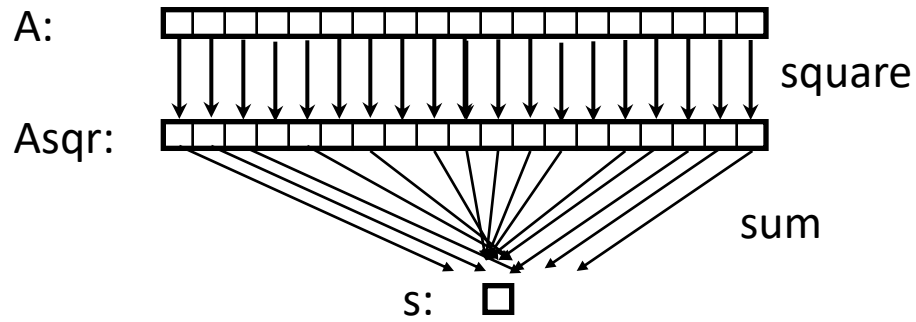
- How can we parallelize this?
- How long will it take if we have unlimited number of processors?

# Example

- First, decompose your problem into a set of tasks
  - There are many ways of doing it.
  - Tasks can be of the same, different, or undetermined sizes.
- Draw a task-dependency graph (do you remember the DAG we saw earlier?)
  - A directed graph with **Nodes** corresponding to tasks
  - **Edges** indicating dependencies, that the result of one task is required for processing the next.



# Example



Does your knowledge of the underlying hardware change your task dependency graph?  
If yes, how?

Where do we lose  
performance?

# Sources of Performance Loss in Parallel Programs

- Extra overhead
  - synchronization
  - communication
- Artificial dependencies
  - Hard to find
  - May introduce more bugs
  - A lot of effort to get rid of
- Contention due to hardware resources
- Coherence
- Load imbalance

# Artificial Dependencies

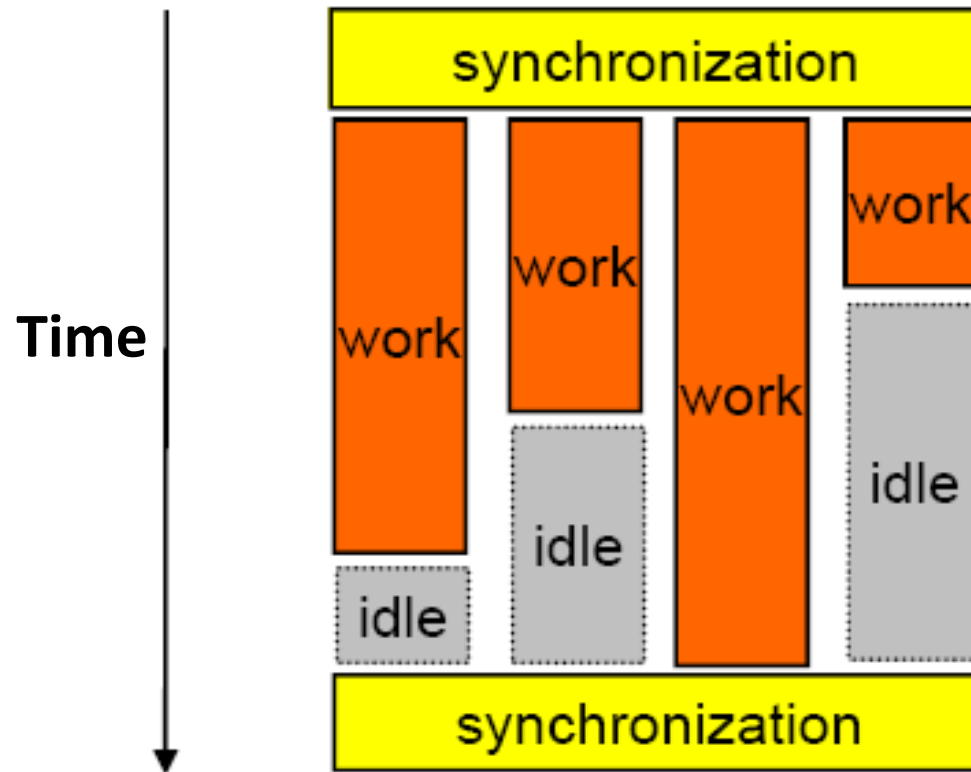
```
int result;  
//Global variable  
  
for (...) // The OUTER loop  
    modify_result(...);  
    if(result > threshold)  
        break;  
  
void modify_result(...)  
    ...  
    result = ...
```

What is wrong with  
that program when  
we try to parallelize  
it?

# Coherence

- Extra bandwidth (scarce resource)
- Latency due to the protocol
- False sharing

# Load Balancing



Load imbalance is more severe as the number synchronization points increases.

# Load Balancing

- If you cannot eliminate it, at least reduce it.
- Static assignment
- Dynamic assignment
  - Has its overhead



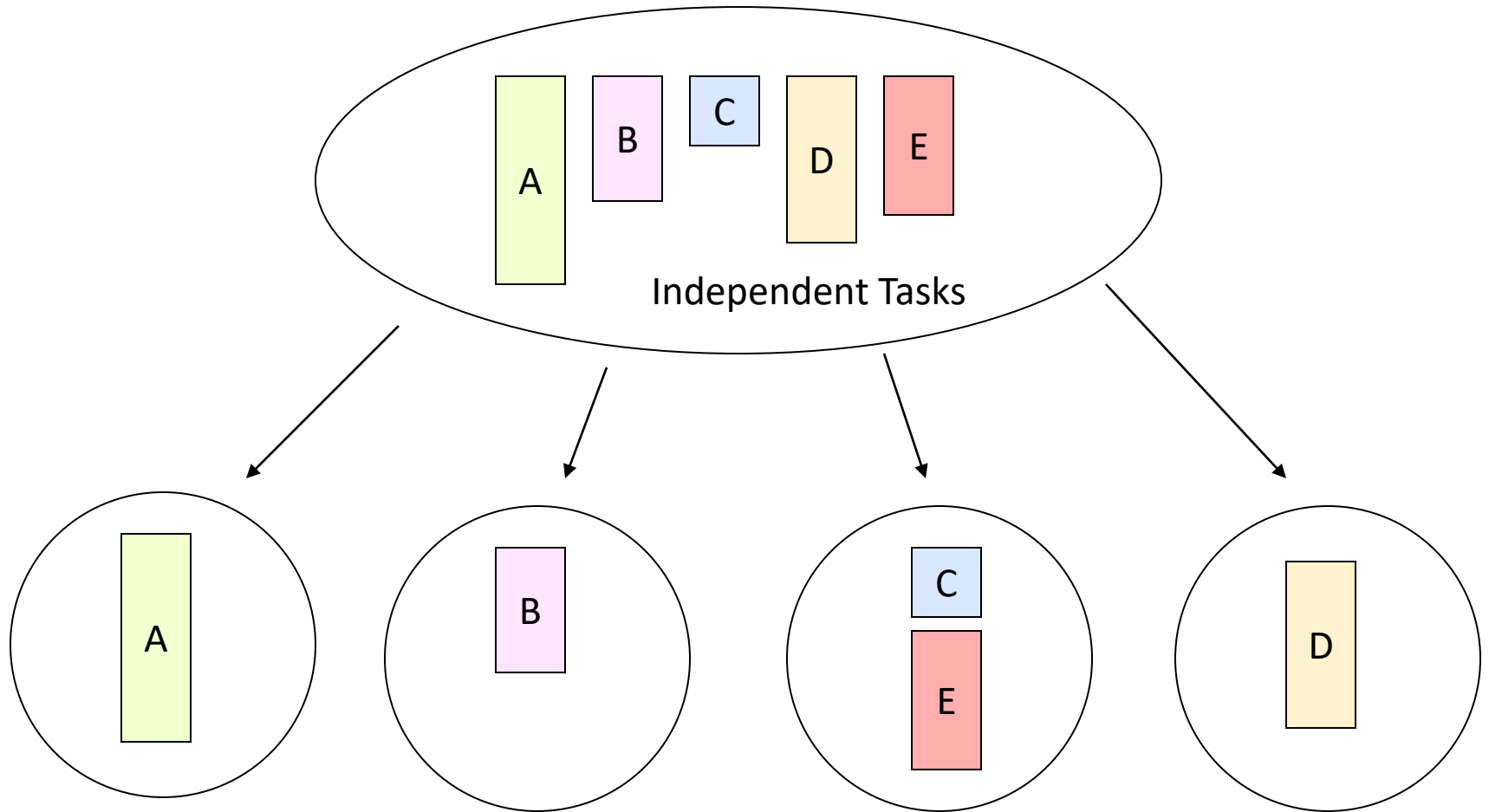
There are several ways for parallelizing an algorithm ... depending on the problem at hand.

What are these ways (or patterns)?

# Patterns in Parallelism

- Task-level (e.g. Embarrassingly parallel)
- Divide and conquer
- Pipeline
- Iterations (loops)
- Client-server
- Geometric (usually domain dependent)
- Hybrid (different program phases)

# Task Level



# Task Level

- Break application into tasks, decided offline (a priori).
- Generally this scheme does not have *strong scalability*.

# Example 1

```
while ( true ) {  
    readUserInput ( ) ;  
    drawScreen ( ) ;  
    playSounds ( ) ;  
    strategize ( ) ;  
}
```



```
Task1 {  
    while ( true ) {  
        readUserInput ( ) ;  
        barrier ( ) ;  
    }  
}
```

```
Task2 {  
    while ( true ) {  
        drawScreen ( ) ;  
        barrier ( ) ;  
    }  
}
```

```
Task3 {  
    while ( true ) {  
        playSounds ( ) ;  
        barrier ( ) ;  
    }  
}
```

```
Task4 {  
    while ( true ) {  
        strategize ( ) ;  
        barrier ( ) ;  
    }  
}
```

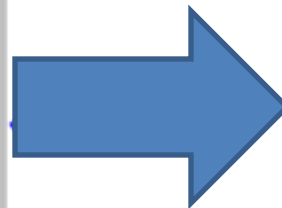
# Example 2

Assume we have a large array and we want to compute its minimum (T1), average (T2), and maximum (T3).

```
#define maxN 1000000000

int m[maxN];
int i;
int min = m[0];
int max = m[0];
double avrg = m[0];

for(i=1; i < maxN; i++) {
    if(m[i] < min)
        min = m[i];
    avrg = avrg + m[i];
    if(m[i] > max)
        max = m[i];
}
avrg = avrg / maxN;
```



```
#define maxN 1000000000
int m[maxN];

int i; int min = m[0];
for(i=1; i < maxN; i++) {
    if(m[i] < min)
        min = m[i];
}

int j;
double avrg = m[0];
for(j=1; j < maxN; j++) {
    avrg = avrg + m[j];
}
avrg = avrg / maxN;

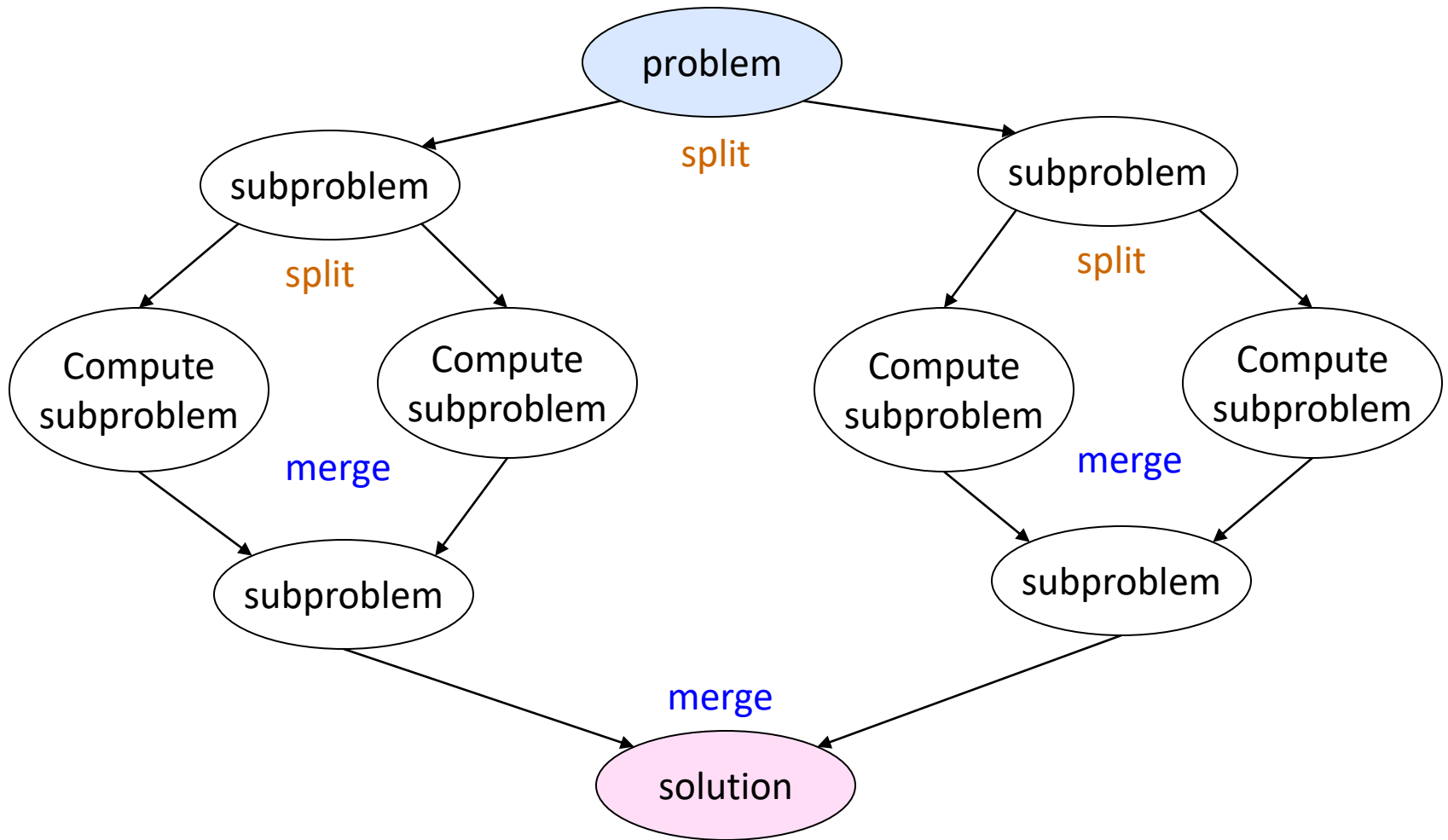
int k; int max = m[0];
for(k=1; k < maxN; k++) {
    if(m[k] > max)
        max = m[k];
}
```

T1

T2

T3

# Divide-And-Conquer



# Divide-And-Conquer

**Sequentially, it looks like this:**

```
// Input: A
DnD ( A ) {
    if ( A is a base case )
        return solution ( A );
    else {
        split A into N subproblems B [ N ];
        for ( int i=0; i<N; i++)
            sol [ i ] = DnD ( B [ i ] ); //
        return mergeSolution ( sol );
    }
}
```



# Divide-And-Conquer

## Parallel Version:

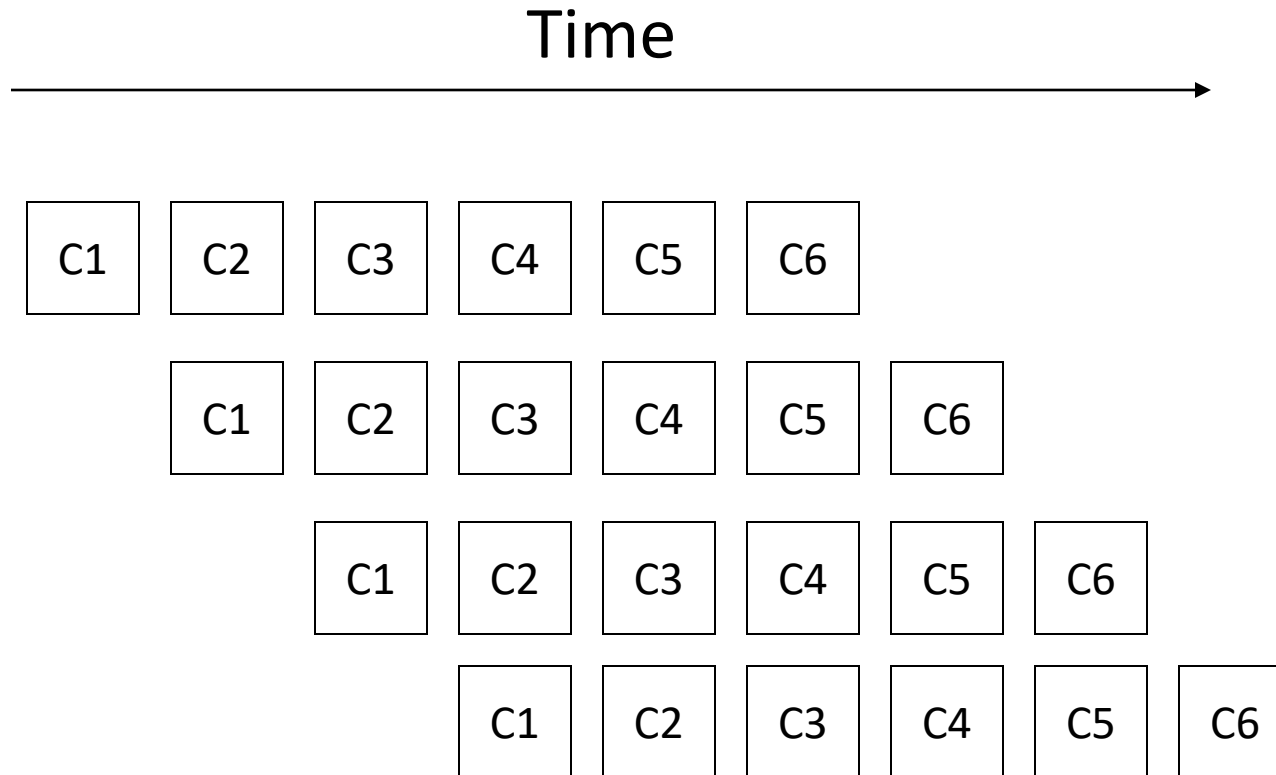
```
// Input: A
DnD(A) {
  if(isBaseCase( A ))
    return solution(A);
  else {
    if( bigEnoughForSplit( A ) ) { // if problem is big enough
      split A into N subproblems B[N];
      for(int i=0;i<N;i++)
        task[i] = newTask( DnD( B[i] ) ); // non-blocking

      for(int i=0;i<N;i++)
        sol[i] = getTaskResult( task[i] ); // blocking results ←
        wait

      return mergeSolution( sol );
    }
    else { // else solve sequentially
      return solution( A );
    }
  }
}
```

# Pipeline

A series of **ordered** but **independent** computation stages need to be applied on data.



# Pipeline

- Useful for
  - streaming workloads
  - Loops that are hard to parallelize
    - due inter-loop dependence
- How to do it?
  1. Split each loop iteration into independent stages (e.g. S1, S2, S3, ...)
  2. Assign each stage to a thread (e.g. T1 does S1, T2 does S2, ...).
  3. When a thread is done with each stage, it can start the same stage for the following loop iteration (e.g. T1 finishes S1 of iteration 0, then start S1 of iteration 1, etc.).
- Advantages
  - Expose intra-loop parallelism
  - Locality increases for variables used across stages
- How shall we divide an iteration into stages?
  - number of stages
  - inter-loop vs intra-loop dependence

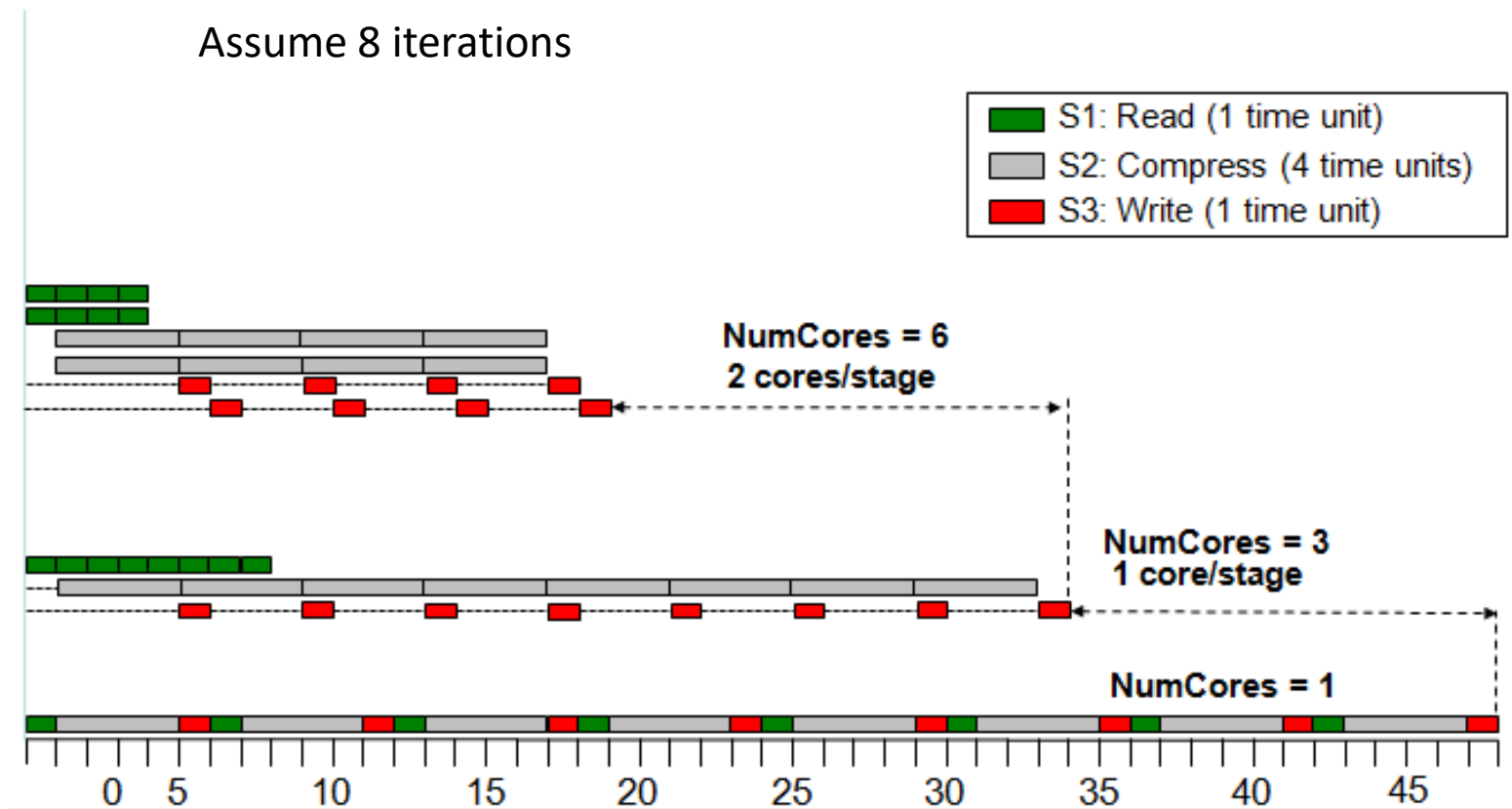
# Example of pipeline parallelism

```
while(!done) {  
    Read block;  
    Compress the block;  
    Write block;  
}
```

Source of example:

<http://www.futurechips.org/parallel-programming-2/parallel-programming-clarifying-pipeline-parallelism.html>

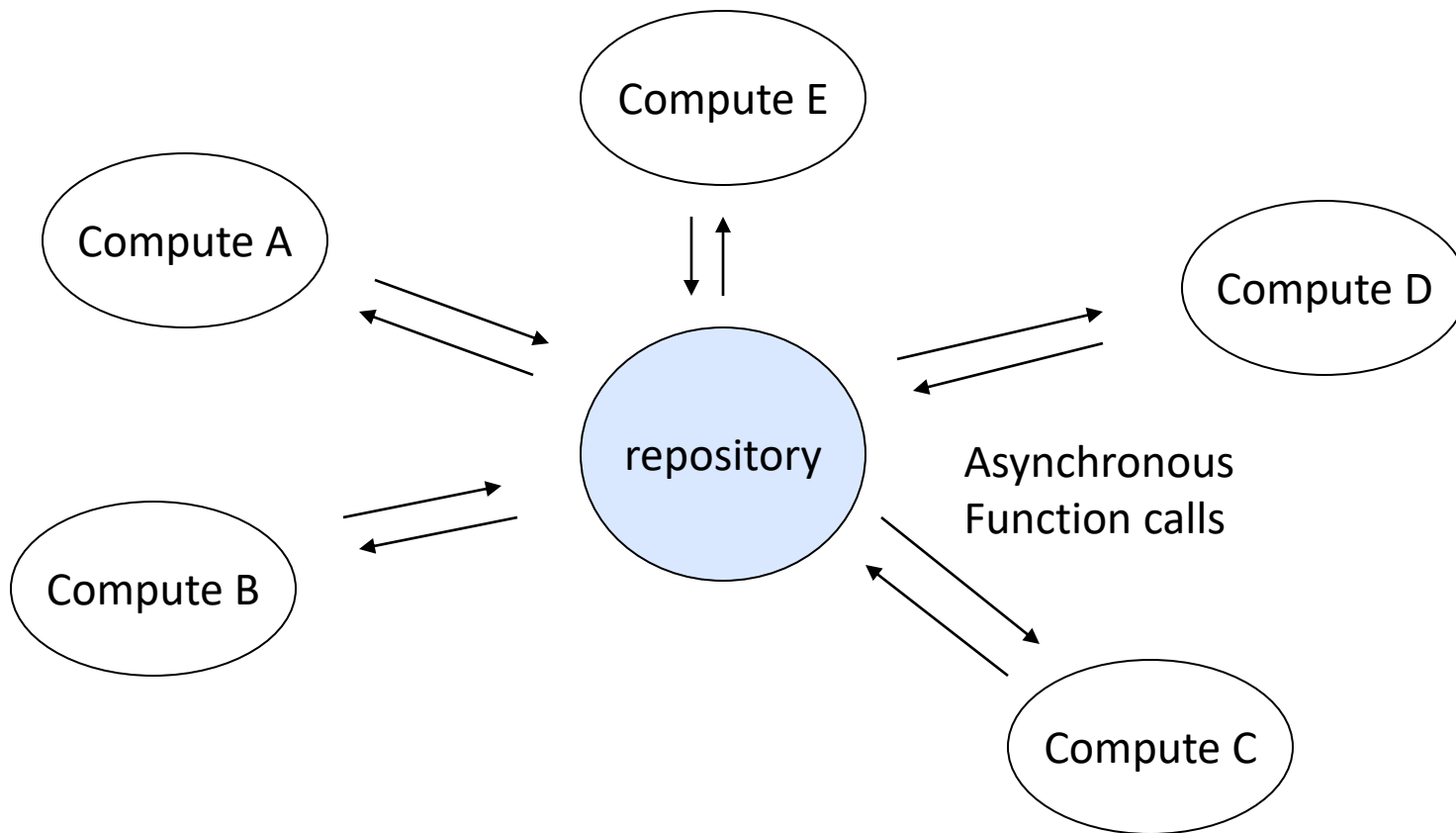
# Example of pipeline parallelism



Source of example:

<http://www.futurechips.org/parallel-programming-2/parallel-programming-clarifying-pipeline-parallelism.html>

# Repository Model



Whenever a thread is done with its task it can take another one from a repository.

# Conclusions

- Concurrency and parallelism are not exactly the same thing.
- **problem → algorithm → dependency graph → parallel pattern → implementation**
- Knowing the hardware will help you generate a better task dependency graph → dependency graph in turn helps you reason about parallelism in your code