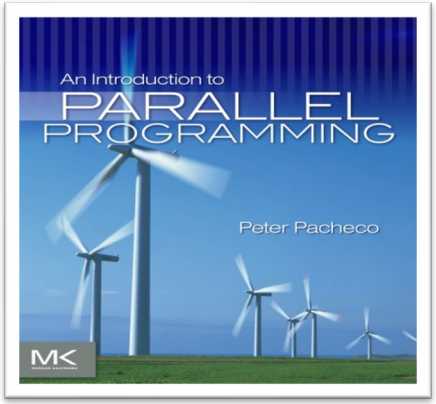
14013204-3 - PARALLEL COMPUTING



Lecture 5

Example problem

- We will parallelize a simple problem

Example Problem: Array Sum

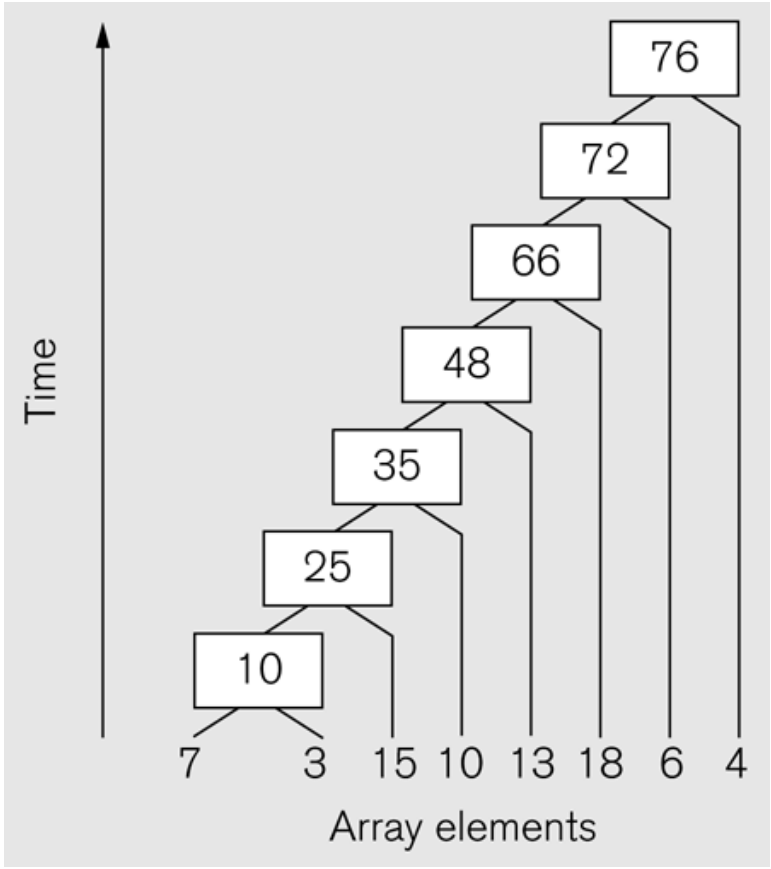
- **Input**: suppose we have an array that has 100 million elements.
  - int size = 100000000;
  - int array[] = {7,3,15,10,13,18,6,4,…};

- **method**: **Add** all the **numbers** in this large array

- What code should we write for a **sequential program**?

Example Problem: Sequential

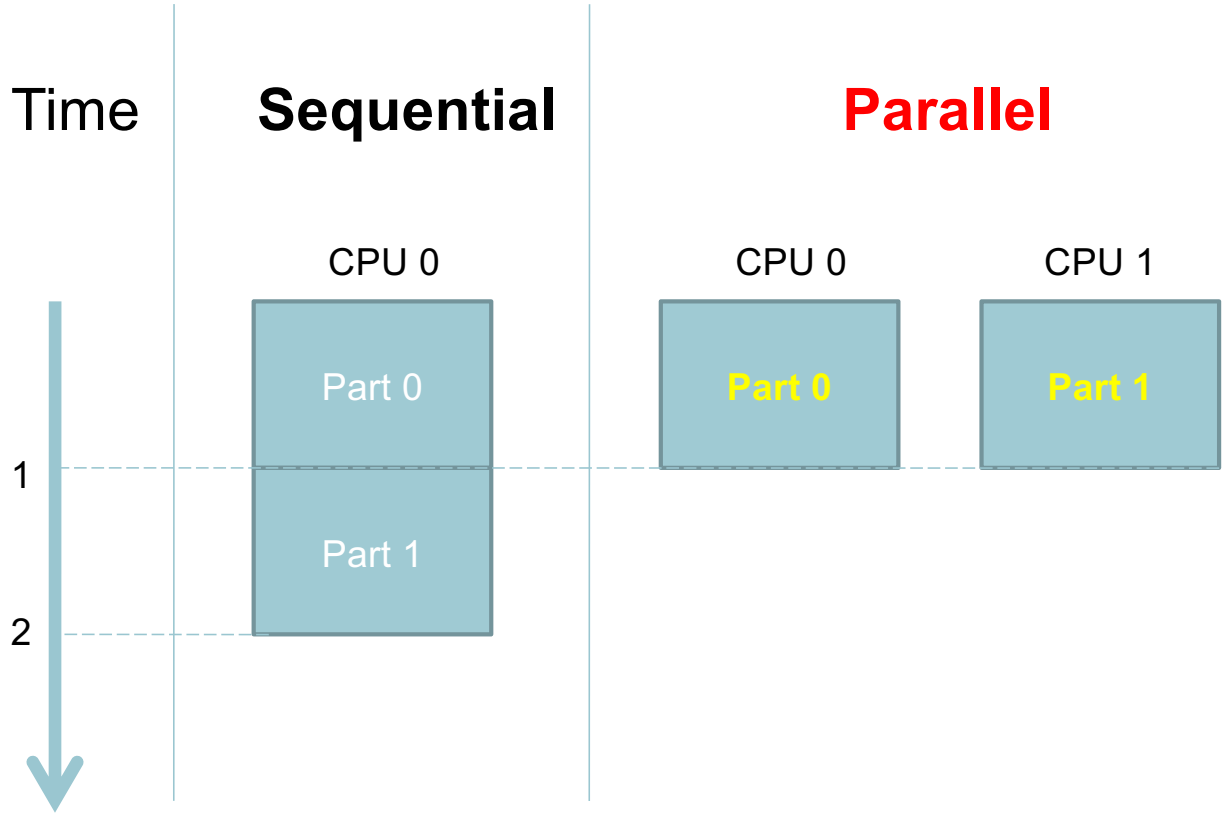
```
int sum = 0;
int i = 0;
for(i = 0; i < size; i++) {
   sum += array[i];
}
```



Parallelizing the sequential code

- **Objective: Thinking about parallelism**
  - **Multiple processors** need something to do
    - A program/software has to be split into parts
    - Each part can be executed on a different processor.
  - **How do we improve performance over a single processor?**
    - If a problem takes 2 seconds on a single processor.
    - And we break it into two (equal) parts: 1 second for each part.
    - We execute the parts separately, but in parallel, on processors, and then we **improve performance**

Parallelizing the sequential code



Parallelizing the sequential code

- **What parts can be done separately?**
  - **Meaning**: What parts have **no data dependence**
  - **Data dependence**:
    - **The execution of an instruction** (or line of code) is **dependent** on execution of a **previous** **instruction.**
  - **Data independence**:
    - The **execution of an instruction** (or line of code) is **not dependent on the execution of a previous instruction.**

Example of Data Dependence

```
int x = 0;
int y = 5;


x = 3;
y = y + x;      //Is this line dependent on the previous line?
```

Data Dependence & Parallelism

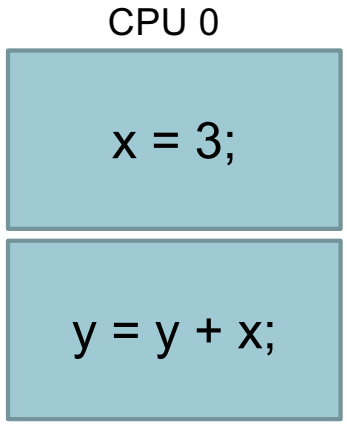
- **In a sequential program:**
  - Data dependence does not matter.
  - Instructions are executed one by one.
- **In a parallel program**
  - **data independence** allows parallel execution of instructions
  - **Data dependence prevents** parallel execution of instructions
    - Reduces parallel performance
    - Reduces the number of processors that can be used

Why is Data Dependence Bad For Parallel Programs?

- **Example: Does not allow correct parallel execution**

```
int x = 0;
int y = 5;

x = 3;
y = y + x;
```



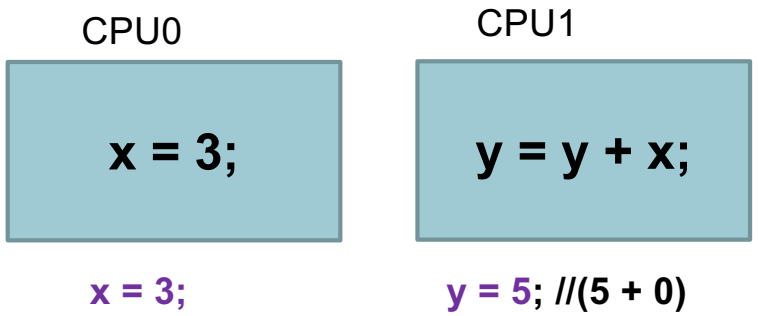
x = 3; y = 8;

Why is Data Dependence Bad For Parallel Programs?

- **Example: Does not allow correct parallel execution**

```
int x = 0;
int y = 5;

x = 3;
y = y + x;
```

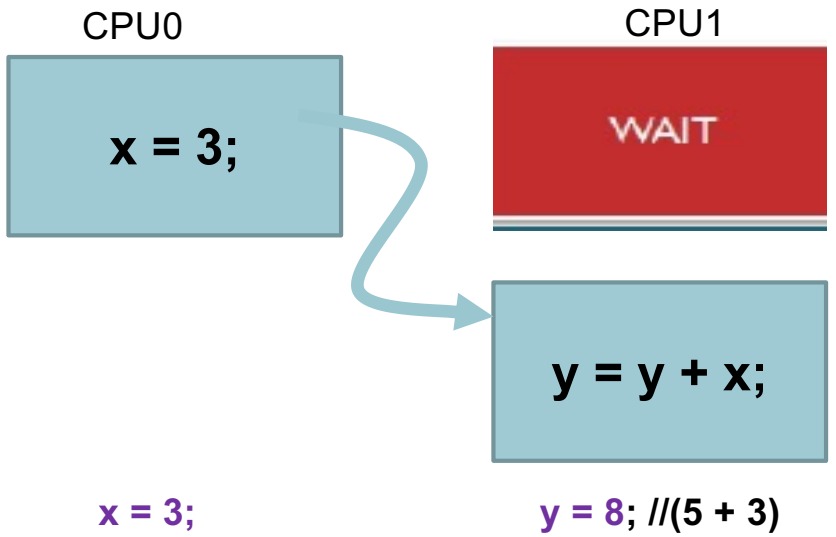


Why is Data Dependence Bad For Parallel Programs?

- **Example: Does not allow correct parallel execution**

```
int x = 0;
int y = 5;

x = 3;
y = y + x;
```

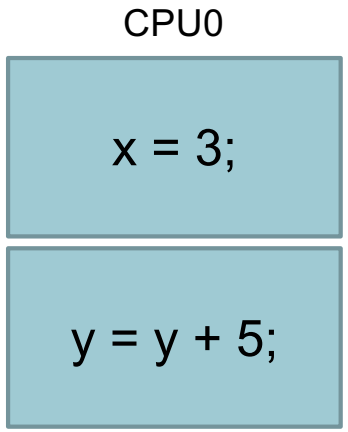


Why is Data Independence Good For Parallel Programs?

- **Example: Allows correct parallel execution**

```
int x = 0;
int y = 5;

x = 3;
y = y + 5;
```



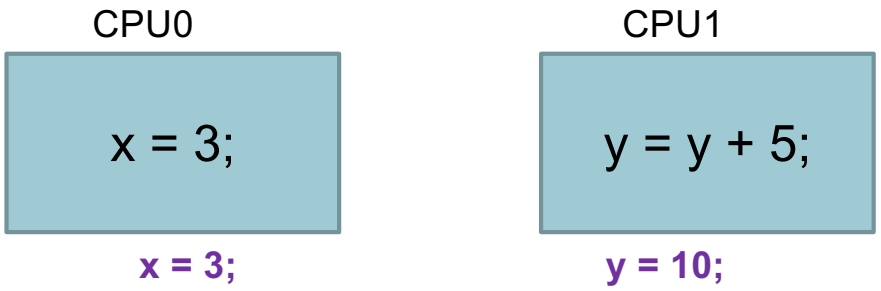
x = 3; y = 10;

Why is Data Independence Good For Parallel Programs?

- **Example: Allows correct parallel execution**

```
int x = 0;
int y = 5;

x = 3;
y = y + 5;
```



Back to Array Sum Example

Does the code have data dependence?

```
int sum = 0;
for(int i = 0; i < size; i++) {
        sum += array[i];      //sum=sum+array[i];
}
```

Not so easy to see

Back to Array Sum Example

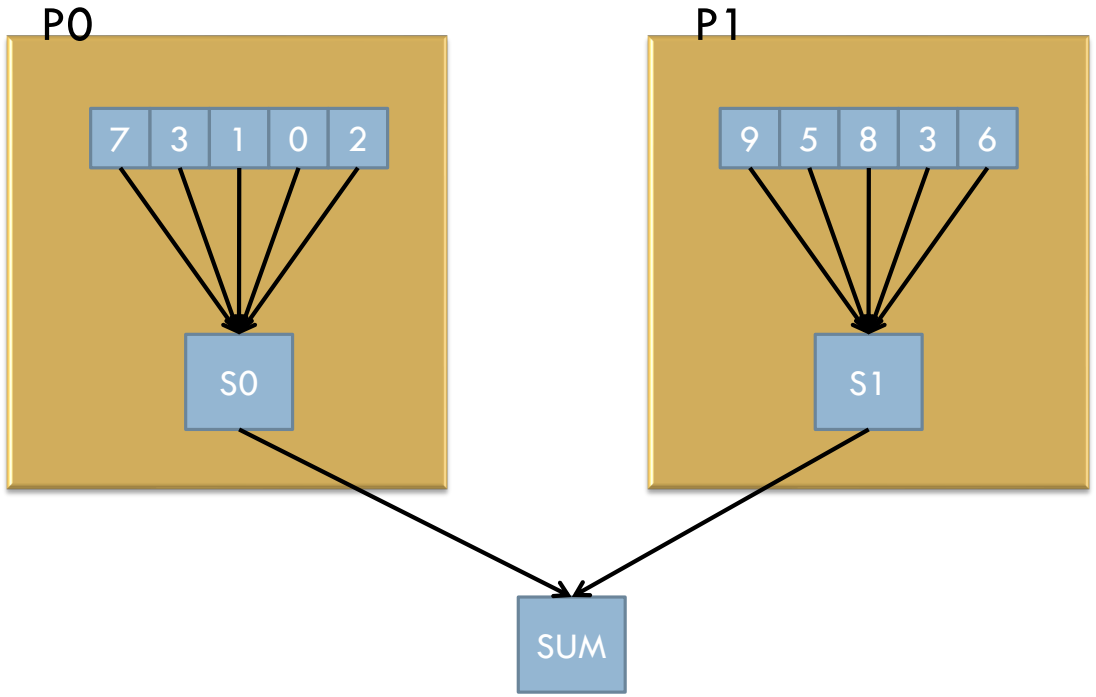
Let's unroll the loop:

```
int sum = 0;
sum += array[0]; //sum=sum+array[0];
sum += array[1]; //sum=sum+array[1];
sum += array[2]; //sum=sum+array[2];
sum += array[3]; //sum=sum+array[3];
…
```

Now we can see dependence!

Removing Dependencies

- Sometimes this is possible.
- Break Sum into Pieces



Some Details: Processors & Processes

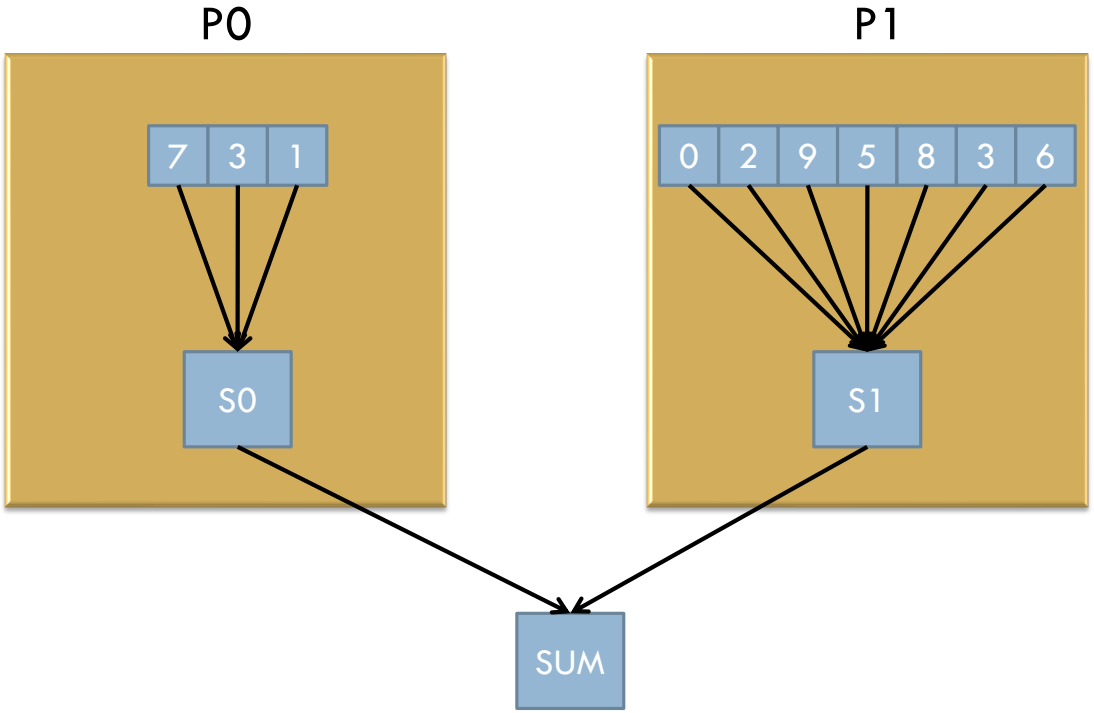
- A program executes inside a process
  - Processor ⇒ hardware level.
  - Process ⇒ software level
- If we want to use multiple processors
  - We need multiple processes
  - One process for each processor
  - Processes are big, heavyweight
- Threads are lighter than processes
  - But same strategy
  - One thread for each processor

What Does the Code Look Like?

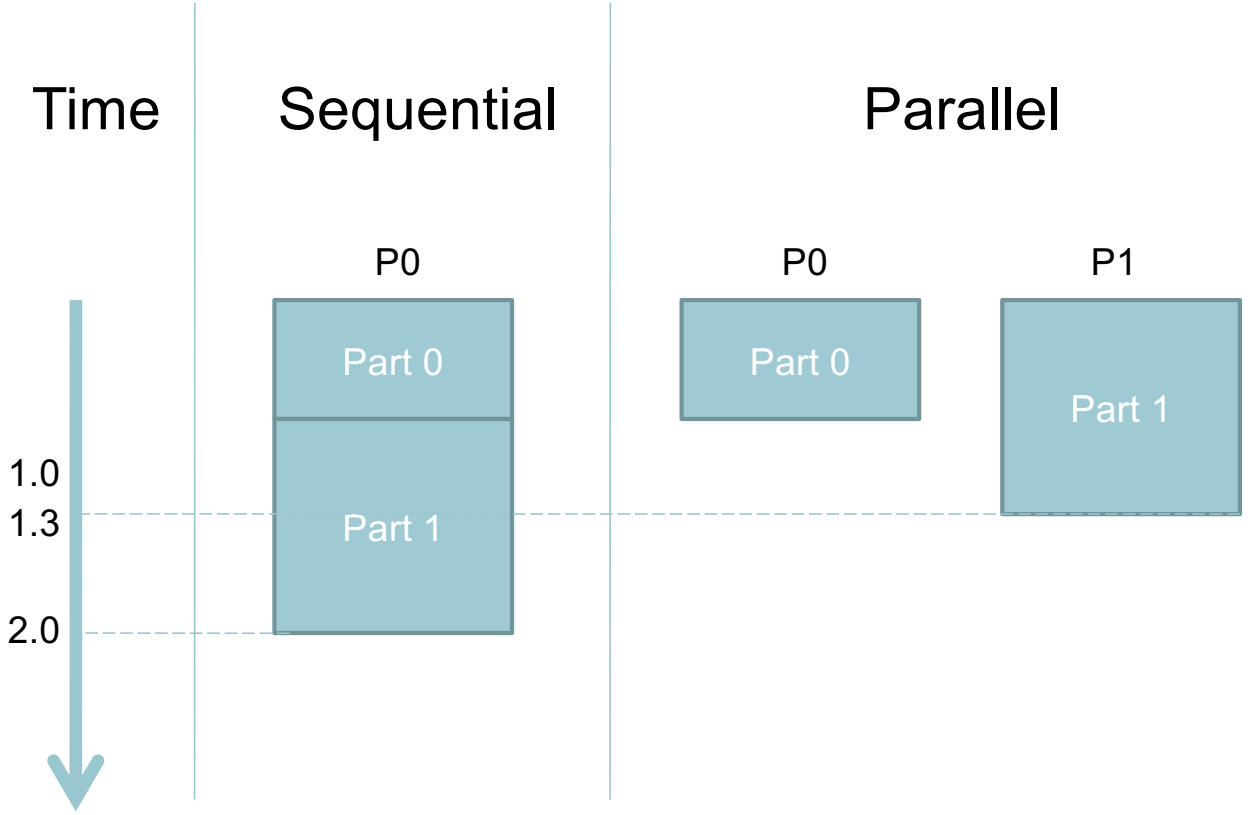
```
int numThreads = 2; //Assume one thread per core, and 2 cores
int sum = 0;
int i = 0;
double middleSum[numThreads];
int threadSetSize = size/numThreads
//Each thread will execute this code with a different threadID
for( i = threadID*threadSetSize; i < (threadID+1)*threadSetSize; i++) {
    middleSum[threadID] += array[i];
}
//Only thread 0 will execute this code
if (threadID==0) {
    for(i = 0; i < numThreads; i++) {
            sum += middleSum[i];
    }
}
```

Load Balancing

- Which processor is doing more work?



Load Balancing



Example Problem: Array Sum

- Parallelized code is more complex
- Requires us to think differently about how to solve the problem
  - Need to think about **breaking it into parts.**
  - Analyze **data dependencies**, and **remove** them if possible
  - Need to **load balance** for **better performance.**

Example Problem: Array Sum

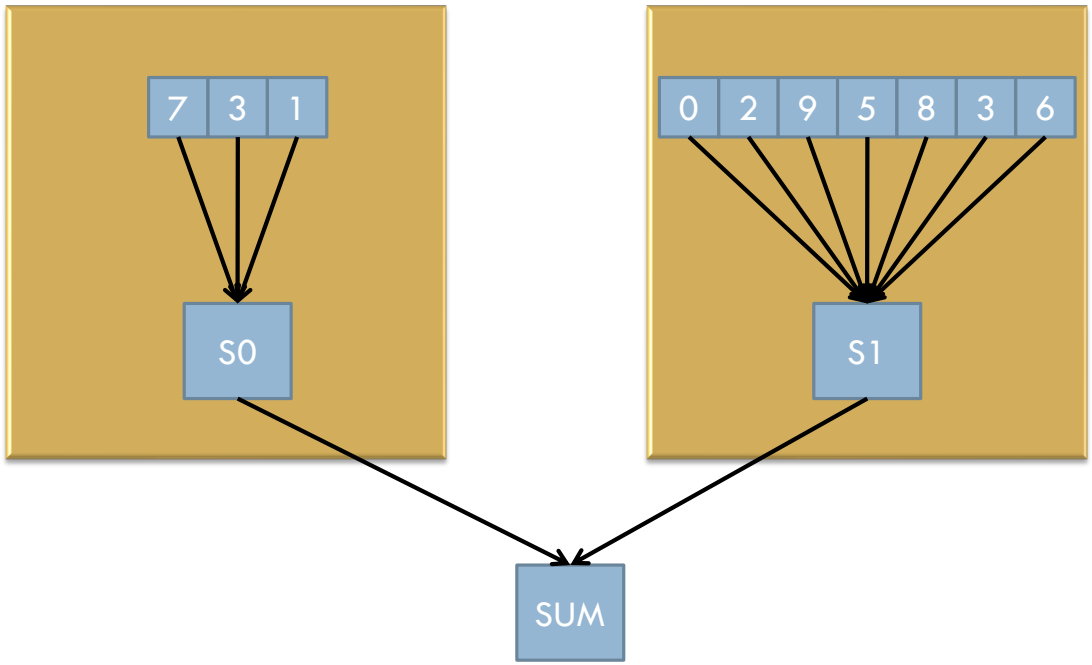
- However, the parallel code is broken
  - Thread 0 adds all the middle sums.
  - What if thread 0 finishes its own work, but other threads have not?

The Parallel Code

```
int numThreads = 2; //Assume one thread per core, and 2 cores
int sum = 0;
int i = 0;
double middleSum[numThreads];
int threadSetSize = size/numThreads
//Each thread will execute this code with a different threadID
for( i = threadID*threadSetSize; i < (threadID+1)*threadSetSize; i++)
{
        middleSum[threadID] += array[i];
}
//Only thread 0 will execute this code but it will not wait for other threads
if (threadID==0) {
    for(i = 0; i < numThreads; i++) {
        sum += middleSum[i];
    }
}
```

Synchronization

- P0 will probably finish before P1



How Can We Fix The Code to GUARANTEE It Works Correctly?

```
int numThreads = 2; //Assume one thread per core, and 2 cores
int sum = 0;
int i = 0;
double middleSum[numThreads];
int threadSetSize = size/numThreads
//Each thread will execute this code with a different threadID
for( i = threadID*threadSetSize; i < (threadID+1)*threadSetSize; i++)
{
        middleSum[threadID] += array[i];
}
//Only thread 0 will execute this code but it will not wait for other threads
if (threadID==0) {
    for(i = 0; i < numThreads; i++) {
        sum += middleSum[i];
    }
}
```

Synchronization

- Sometimes we need to **coordinate/organize threads.**
- If we don't, the code might calculate the wrong answer to the problem.

- Can happen even if load balance is perfect.

- **Synchronization** is concerned with this **coordination/organization.**

Can with Synchronization Fixed?

```
int numThreads = 2; //Assume one thread per core, and 2 cores
int sum = 0;
int i = 0;
double middleSum[numThreads];
int threadSetSize = size/numThreads
//Each thread will execute this code with a different threadID
for( i = threadID*threadSetSize; i < (threadID+1)*threadSetSize; i++)
{
        middleSum[threadID] += array[i];
}
WaitForAllThreads(); // wait for all threads
//Only thread 0 will execute this code
if (threadID==0) {
    for(i = 0; i < numThreads; i++) {
        sum += middleSum[i];
    }
}
```

Synchronization

- The example shows a **barrier**
  - This is **one type of synchronization**.

- **Barriers require all threads** to reach that point in the code before any thread is allowed to continue.

- **It is like a gate**. All threads come to the gate, and then it opens.

Synchronization

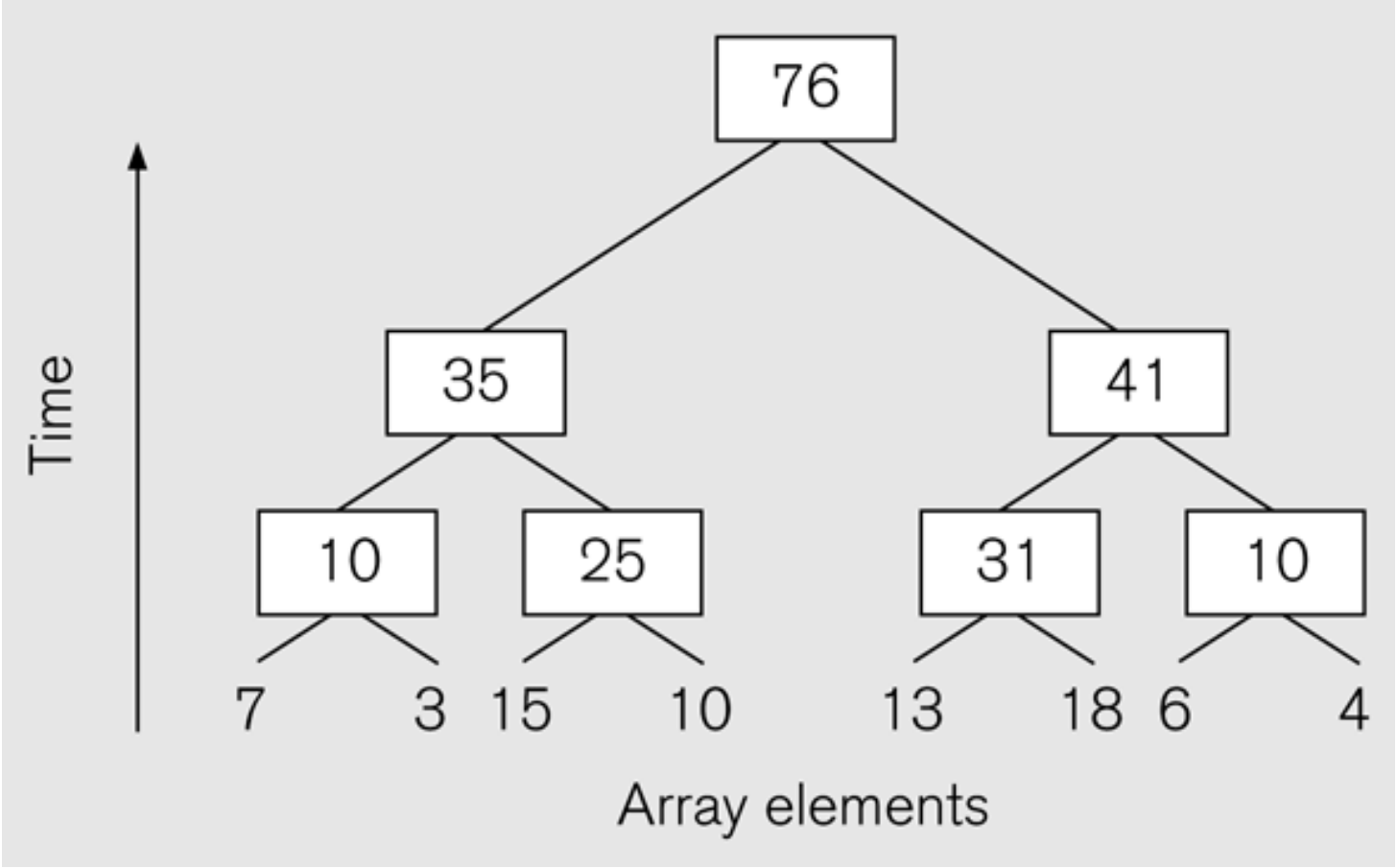
- This is one type of synchronization used to **avoid the problem of race conditions**.
- Before adding the barrier, the code result was **non-deterministic.**
- The result now is **deterministic.**

- **Terminology:**
  - **race condition**: When threads or processes attempt to simultaneously access a resource, and the accesses can result in an error.
  - **Non-determinism**: A computation is nondeterministic if a given input can result in different outputs. If multiple threads are executing independently, the relative rate at which they'll complete statements varies from run to run, and hence the results of the program may be different from run to run.

Generalizing the Solution

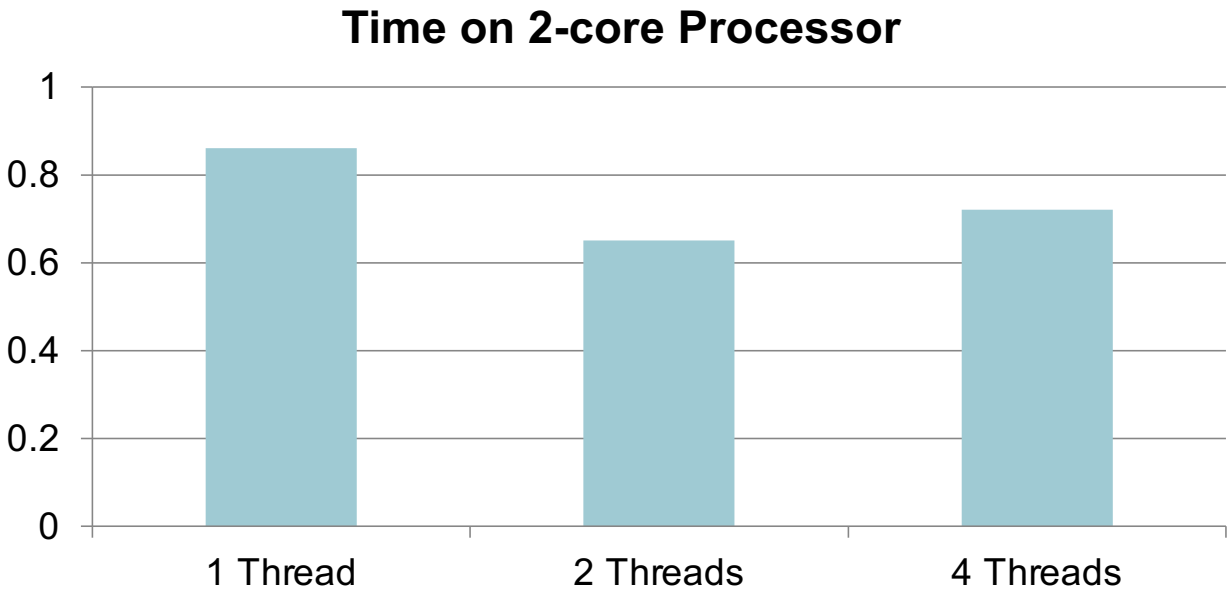
- We only looked at **how to parallelize for 2 threads**
- But the **code is more general**
  - Can use **any number of threads.**

Parallel Program



Performance

- Now the program is correct
- Let's look at performance



Performance

- **Two-threads are not 2x fast**. **Why**?
  - The problem is called **false sharing.**
  - We discussed this in the previous lecture.
  - Note: Unlike race condition, false sharing does not affect the correctness of the parallel code. its impact is only **in performance.**

- **Four threads slower than two threads**. Why?
  - **The processor only has two cores**
  - **Four** threads **add scheduling overhead**, and waste time.

False sharing in the code

- There is a potential for false sharing on array middleSum.
- Cache lines are a power of 2 of contiguous bytes which are typically 32-256 in size. The most common cache line size is 64 bytes.
- This array is partitioned according to the number of threads.
- Each component of the array middleSum will be stored in 8 bytes, so the array components fit in the same cache line.
- When executed in parallel, the threads modify different but adjacent, elements of middleSum, which invalidates the cache line for all processors.

- The frequent coordination required between processors when cache lines are marked Invalid requires cache lines to be written to memory and subsequently loaded.

Always remember

- In **sequential programs** we focus only on **computation**

- In the **parallel program** we focus on **computation + coordination**
- **Coordination involves different aspects:**
  - **Divide the work** among the available cores.
  - **Assign the work** to cores.
  - **Balance the load** to keep all cores busy during the program execution as much as possible.
  - **Manage synchronization** in shared memory and **communication** in distributed memory.
  - **Identify and reduce different sources of performance** issues such as **data dependency**, **false sharing**, **data locality**, etc.

Summary

- Discussed an example to understand how to parallelize code and some of the main related issues
  - Data dependency
  - Load balancing
  - Synchronization

- Each will be discussed in more detail in later lectures