Parallel Algorithms I

Outline

Thinking in Parallel

Embarrassingly Parallel Algorithms

- **Example #1: Vector Addition**
- **Example #2: Summation**
- **Example #3: Random Number Generation**

Why pursue parallelized algorithms?

Thinking in Parallel

To become a **parallel programmer**, a key skill is *understanding how to turn a sequential algorithm into a parallel algorithm*

Some questions to ask when considering moving to a parallel solution to a problem:

- *Could this program really benefit from being parallelized?*
- *What kind of performance increase might be possible?*
- *How easy is it to turn this implementation into a parallel one?*
- *Which parts of the code can be parallelized?*

Embarrassingly Parallel Algorithms

We start this journey to parallel thinking by examining a few algorithms known as **embarrassingly parallel algorithms**

As the name implies, these are programs *where it takes extremely little effort (little or no modification) to develop a parallel solution*

- In terms of the individual running computations, it usually means they can work independently with little or no communication needed

Before getting to more challenging examples, it helps our understanding of parallelization to start here

Example #1: Vector Addition

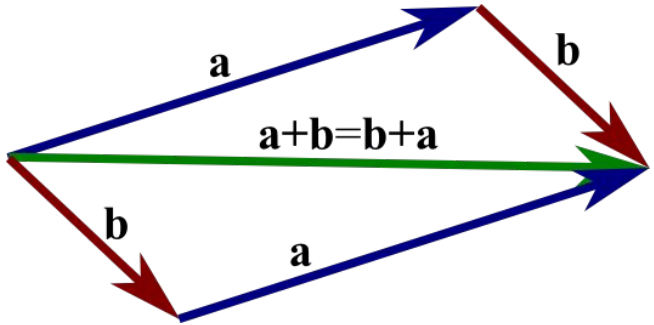
An extremely common mathematical operation is taking two vectors, and simply adding them together

Some examples:

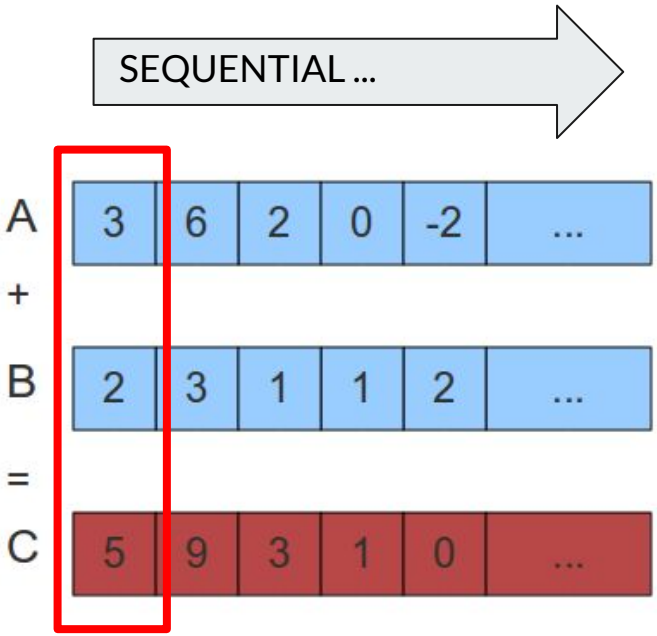
(1, 5) + (2, 3) = (3, 8)

(3, 2, 1, 0) + (5, 4, 5, 7) = (8, 6, 6, 7)

Or graphically, adding vectors looks something like this:



Example #1: Vector Addition



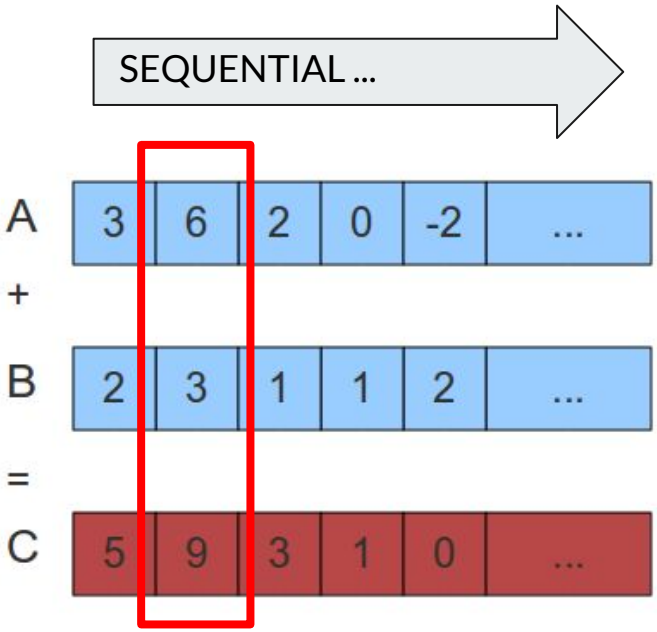
The **sequential solution**:

Loop over each corresponding position in the two vectors from left to right, adding the two values together and storing the result in the same position of a new result vector

Assume you have a second compute core. ***How could you develop a solution where both cores work together?***

- If it helps, think of the two cores as two people, and assume the vectors you are adding together each have 100 numbers. How could you speed up this process? What if you had even more cores?

Example #1: Vector Addition



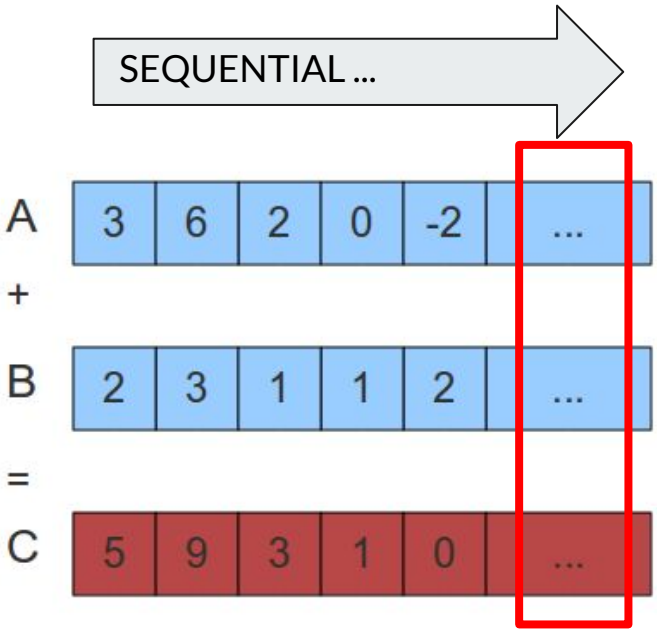
The **sequential solution**:

Loop over each corresponding position in the two vectors from left to right, adding the two values together and storing the result in the same position of a new result vector

Assume you have a second compute core. ***How could you develop a solution where both cores work together?***

- If it helps, think of the two cores as two people, and assume the vectors you are adding together each have 100 numbers. How could you speed up this process? What if you had even more cores?

Example #1: Vector Addition



The **sequential solution**:

Loop over each corresponding position in the two vectors from left to right, adding the two values together and storing the result in the same position of a new result vector

Assume you have a second compute core. ***How could you develop a solution where both cores work together?***

- If it helps, think of the two cores as two people, and assume the vectors you are adding together each have 100 numbers. How could you speed up this process? What if you had even more cores?

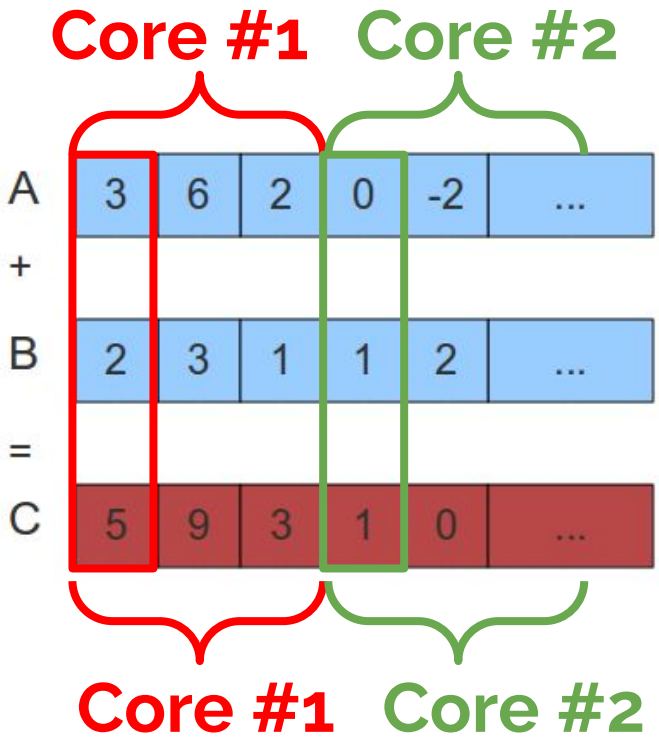
Example #1: Vector Addition

The **parallel solution**:

Do the same thing as the sequential solution, but have the first core work on half the positions, and the second core work on the other half.

More generally: For `n` cores, each core could be assigned `1/n` of the numbers

Notice that **no communication** would be needed between the cores, other than deciding which core works on which numbers at the very beginning. Remember that communication is **SLOW**!



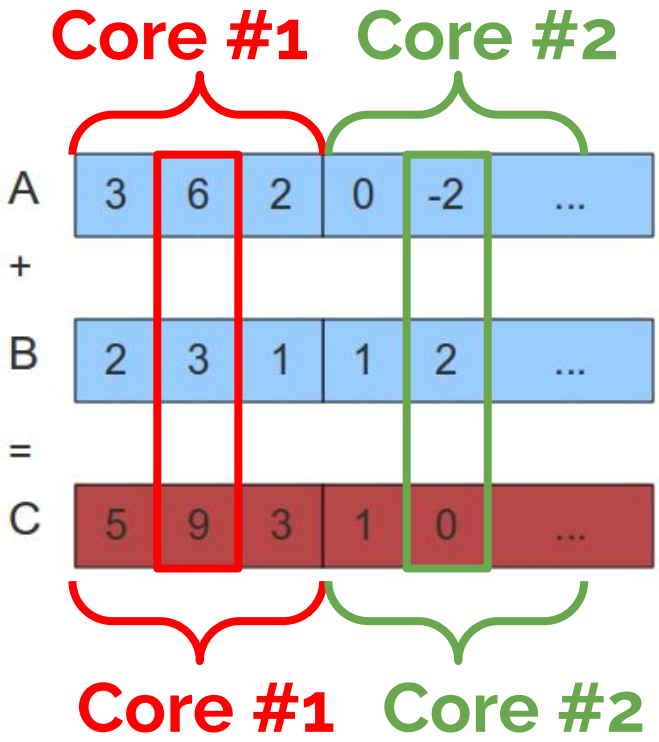
Example #1: Vector Addition

The **parallel solution**:

Do the same thing as the sequential solution, but have the first core work on half the positions, and the second core work on the other half.

More generally: For `n` cores, each core could be assigned `1/n` of the numbers

Notice that **no communication** would be needed between the cores, other than deciding which core works on which numbers at the very beginning. Remember that communication is **SLOW**!



Example #2: Summation

Another common mathematical operation is adding up all the numbers in a list

Some examples:

sum([1, 4, 7]) = 12

sum([3, 2, 1, 0, 5, 4, 5, 7]) = 27

This is problem is *similar* to the first example, with an important difference that we will see shortly.

Example #2: Summation

sum([3, 2, 1, 0, 5, 4, 5, 7]) = 27

The **sequential solution**:

Loop over each number from left to right, adding each number into a summation variable that has been initialized to 0

Sequential Algorithm:

running sum + new number = new sum

0 + 3 = 3 → 3+2=5→ 5+1=6→

6+0=6→ 6+5=11→ 11+4=15→

15+5=20→ 20+7=27

Again assume you have a second compute core. ***How could you develop a solution where both cores work together?***

- If it helps, think of the two cores as two people, and assume the list has 100 numbers in it. How could you speed up this process? What if you had even more cores?

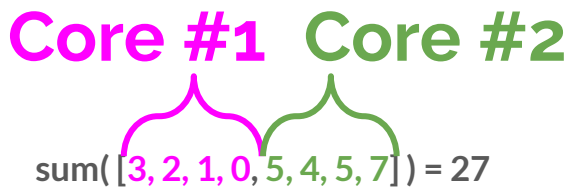
Example #2: Summation

The **parallel solution**:

Do the same thing as the sequential solution, but have the first core work on half the positions, and the second core work on the other half. Each core will keep a separate variable with the values that it has summed up so far (a **partial sum**). **At the end, all the partial sums are added together.**

More generally: For `n` cores, each core could be assigned `1/n` of the numbers (with `n` partial sums).

Again, communication is minimal. You would have to decide which core works on which numbers at the very beginning, and each core would have to communicate its partial sum in the end.



Example #3: Random Number Generation

Many algorithms use a technique called **Monte Carlo Simulation**, which really just means that random numbers are used in some way.

One example of a straightforward random number generator is called a **linear congruential generator**.

Often Based on Date/Time Stamp

Given an initial `seed` for the first random number (r_0), as well as fixed values for **a**, **b**, and **c**, you can generate a new random number in the range 0 to (**c**−1) by using the formula:

$$r_i = (r_{i-1} * a + b) \bmod c$$

recursive!!

Monte Carlo, Monaco
(The "Las Vegas" of Europe)

Example #3: Random Number Generation

For example, assume that you have the following:

seed = 32

a = 12

b = 51

c = 101

You will generate the following list of numbers:

32, 31, 19, 77, 66, 35, 67, 47, 9, 58, ...

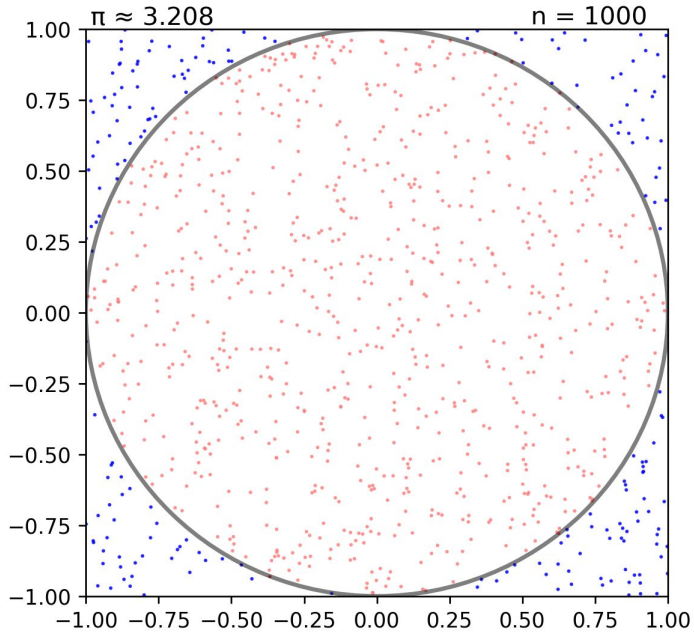
seed (r_0)

Example #3: Random Number Generation

It is straightforward to implement a sequential program to compute these numbers using a loop.

There are many problems that use this Monte Carlo technique, such as ***estimating the number pi*** and *estimating the area under a curve*.

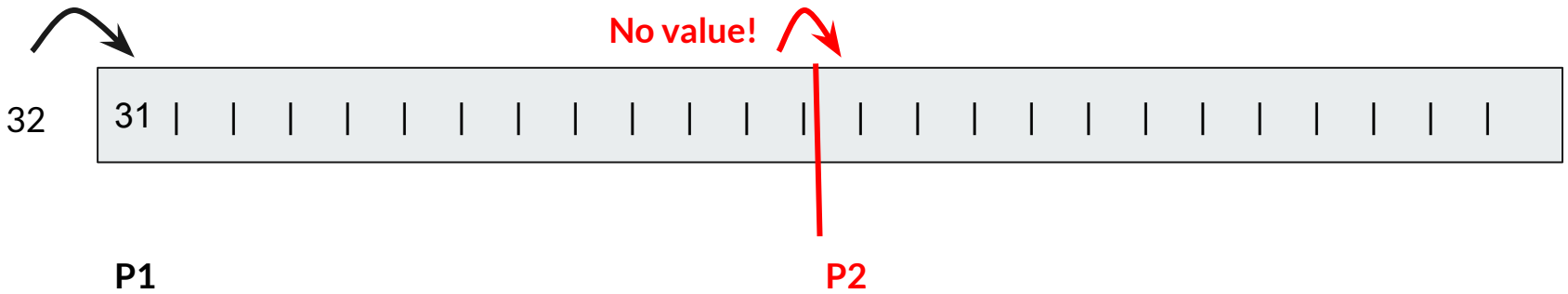
Assume that you have to generate 10 million random numbers. **Can this be parallelized to speed up the process?**



Example #3: Random Number Generation

We could first try the previous approach of just letting Process 1 generate the first half of the numbers, while Process 2 generates the second half.

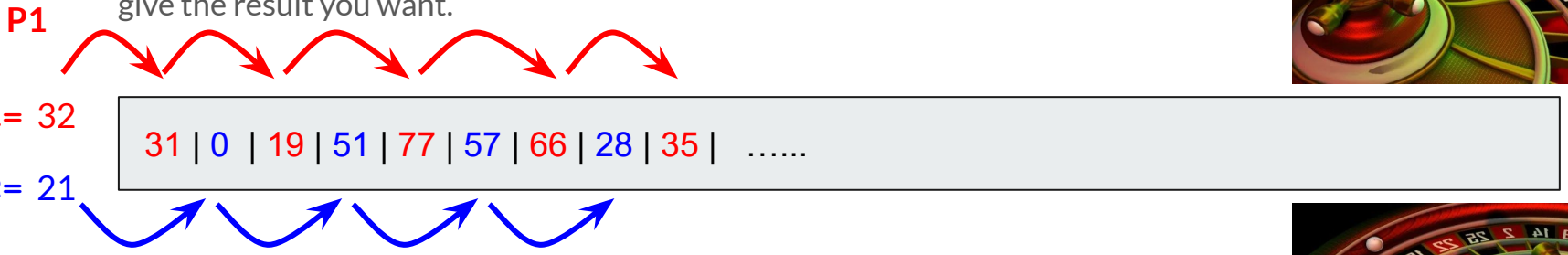
The problem? Any given number in the sequence ***requires*** that the previous number is already there.



Example #3: Random Number Generation

One solution: each process could use a different `seed` value (and possibly a different `a` and `b` value) to generate alternating numbers in the sequence (technically *two* ***completely separate*** *generators*).

This will make the two processes completely independent of each other, and still give the result you want.



Why Pursue Parallel Algorithms?

Faster runtime

- More cores theoretically means that work will get done faster (*speedup*)

Greater scaling

- More cores can be used to model bigger problems in the same amount of time as fewer cores can model smaller problems

Better accuracy

- When more cores are used, there is more computing power available for error checking, so the final result should be a better approximation