Parallel Programming
Programowanie równoległe

Lecture 4: Parallel algorithms.

Paweł Rzążewski

RAM model

When designing and analysing sequential algorithms, we usually use the so-called *RAM* (Random Access Machine) model.
We have one processor, executing the commands sequentially, one by one. Each operation (arithmetic operation, memory read/write etc.) takes unit time.
We are interested in the complexity (optimistic, pessimistic, average), which is the number of operations performed by the algorithm (equal to the computation time).

PRAM model

In parallel algorithms, the model we usually use is *PRAM* (Parallel RAM).

We have a number of processors, working in parallel, in synchronized way (the processors execute one step at the same time).

The processors share a global memory (which is also used for the synchronization). Each processor knows its index. Also, we assume that the total number of processors in known.

PRAM model – continued

The parallel parts of the algorithm are denoted by the `parallel for` (sometimes we will write `parallel for all`).
So, when we write:

```
parallel for all i ∈ X do
  action(i)
```

we actually mean that for each element i of X we assign one processor and this processor executes the method `action` on its own element.

Mean square error

We have n observations of the same value ($X = [x_1, x_2, \ldots, x_n]$).
Our goal is to compute the mean-square error E.

Mean square error

We have n observations of the same value ($X = [x_1, x_2, \ldots, x_n]$).
Our goal is to compute the mean-square error E.
The average value is $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$. What is interesting for us is

$$
\begin{aligned}
E &= \sum_{i=1}^{n} (x_i - \bar{x})^2 \\
&= \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} 2\bar{x}x_i + \sum_{i=1}^{n} \bar{x}^2 \\
&= \sum_{i=1}^{n} x_i^2 - 2\bar{x} \sum_{i=1}^{n} x_i + n \cdot \bar{x}^2 \\
&= \sum_{i=1}^{n} x_i^2 - 2n \cdot \bar{x}^2 + n \cdot \bar{x}^2 \\
&= \sum_{i=1}^{n} x_i^2 - n \cdot \bar{x}^2.
\end{aligned}
$$

Mean square error – first approach

Consider the following (stupid) algorithm.

Algorithm: MSE1

$X2[\ldots] \leftarrow$ vector of length n
$E \leftarrow 0$
$sx \leftarrow 0$
$sx2 \leftarrow 0$
parallel foreach $i = 1, 2, \ldots, n$ **do**
    $X2[i] \leftarrow X[i] \cdot X[i]$
foreach $i = 1, 2, \ldots, n$ **do**
    $sx \leftarrow sx + X[i]$
    $sx2 \leftarrow sx2 + X2[i]$
$E \leftarrow sx2 - sx \cdot sx/n$
return E

Complexity of parallel algorithms

When estimating the complexity, we have to differentiate between the computation time and the number of operations.

Time complexity

Time complexity measures the computation time. The complexity of each parallel loop is the maximum complexity of each task performed in parallel.

Work complexity

Work complexity measures the total number of operations executed during the computation. This is the complexity of the algorithm executed on single-processor machine.

Complexity of parallel algorithms

We say that a parallel algorithm is sequentially optimal if its work complexity is equal to the complexity of the best sequential algorithm for this problem.

Complexity of parallel algorithms

We say that a parallel algorithm is sequentially optimal if its work complexity is equal to the complexity of the best sequential algorithm for this problem.

Often sequential optimality requires lots of synchronization, which destroys parallelism (and thus the time complexity).

EREW, CREW, CRCW

There are several different submodels of PRAM. The most important are:

- **CREW** (Concurrent Read, Exclusive Write) many processors may read the same memory cell at the same time, only one may write to it.
- **EREW** (Exclusive Read, Exclusive Write) at most one process may access a single memory cell, either for reading or writing,
- **CRCW** (Concurrent Read, Concurrent Write) many processes may write in a single memory cell at the same time.

The most common model is CREW. However, they are essentially different.

Concurrent write

If we allow for a concurrent write, we have to specify the behavior in case of conflicts. The common behaviors are:

- the result is any of the values written in the common cell,
- the result is the value written by the processor with the smallest id,
- the result is some function (e.g. sum) of the values written.

A typical trick in the design of algorithms is to make sure that if many processors write in the shared memore cell (in parallel), then every process writes the same value.

Find one in CRCW PRAM

Suppose we have a binary vector X of length n. Our goal is to report, if there exists at least one 1 in X.
A sequential algorithm has complexity

Find one in CRCW PRAM

Suppose we have a binary vector X of length n. Our goal is to report, if there exists at least one 1 in X.
A sequential algorithm has complexity $O(n)$.

Find one in CRCW PRAM

Suppose we have a binary vector X of length n. Our goal is to report, if there exists at least one 1 in X.
A sequential algorithm has complexity $O(n)$.
Consider the following algorithm on CRCW PRAM.

Algorithm: FindOne-CRCW

result $\leftarrow 0$
parallel for $i = 1, 2, \ldots, n$ **do**
  if $X[i] = 1$ **then** *result* $\leftarrow 1$;
return *result*

Find one in CRCW PRAM

Suppose we have a binary vector X of length n. Our goal is to report, if there exists at least one 1 in X.

A sequential algorithm has complexity $O(n)$.

Consider the following algorithm on CRCW PRAM.

Algorithm: FindOne-CRCW

$\mathit{result} \leftarrow 0$

parallel for $i = 1, 2, \ldots, n$ **do**

  if $X[i] = 1$ **then** $\mathit{result} \leftarrow 1$;

return *result*

Number of processors used:

Find one in CRCW PRAM

Suppose we have a binary vector X of length n. Our goal is to report, if there exists at least one 1 in X.
A sequential algorithm has complexity $O(n)$.
Consider the following algorithm on CRCW PRAM.

Algorithm: FindOne-CRCW

$result \leftarrow 0$
parallel for $i = 1, 2, \ldots, n$ **do**
  if $X[i] = 1$ **then** $result \leftarrow 1$;
return $result$

Number of processors used: n
Time complexity:

Find one in CRCW PRAM

Suppose we have a binary vector X of length n. Our goal is to report, if there exists at least one 1 in X.
A sequential algorithm has complexity $O(n)$.
Consider the following algorithm on CRCW PRAM.

Algorithm: FindOne-CRCW

```
result ← 0
parallel for i = 1, 2, ..., n do
    if X[i] = 1 then result ← 1;
return result
```

Number of processors used: n
Time complexity: $O(1)$
Work complexity:

Find one in CRCW PRAM

Suppose we have a binary vector X of length n. Our goal is to report, if there exists at least one 1 in X.
A sequential algorithm has complexity $O(n)$.
Consider the following algorithm on CRCW PRAM.

Algorithm: FindOne-CRCW

result $\leftarrow 0$
parallel for $i = 1, 2, \ldots, n$ **do**
  if $X[i] = 1$ **then** *result* $\leftarrow 1$;
return *result*

Number of processors used: n
Time complexity: $O(1)$
Work complexity: $O(n)$

Find max in CRCW PRAM

Suppose we have a vector X of length n. Our goal is to find the index of the maximum value in X.

A sequential algorithm has complexity

Find max in CRCW PRAM

Suppose we have a vector X of length n. Our goal is to find the index of the maximum value in X.
A sequential algorithm has complexity $O(n)$.

Find max in CRCW PRAM

Suppose we have a vector X of length n. Our goal is to find the index of the maximum value in X.

A sequential algorithm has complexity $O(n)$.

Algorithm: FindMax-CRCW

```
M ← vector of length n
parallel for 1 ≤ i ≤ n do
    M[i] ← 1
parallel for 1 ≤ i < j ≤ n do
    if X[i] < X[j] then M[i] ← 0;
parallel for 1 ≤ i ≤ n do
    if M[i] = 1 then result ← i;
return result
```

Find max in CRCW PRAM

Suppose we have a vector X of length n. Our goal is to find the index of the maximum value in X.
A sequential algorithm has complexity $O(n)$.

Algorithm: FindMax-CRCW

$M \leftarrow$ vector of length n
parallel for $1 \leq i \leq n$ **do**
  $M[i] \leftarrow 1$
parallel for $1 \leq i < j \leq n$ **do**
  if $X[i] < X[j]$ **then** $M[i] \leftarrow 0$;
parallel for $1 \leq i \leq n$ **do**
  if $M[i] = 1$ **then** $result \leftarrow i$;
return $result$

Number of processors used:

Find max in CRCW PRAM

Suppose we have a vector X of length n. Our goal is to find the index of the maximum value in X.
A sequential algorithm has complexity $O(n)$.

Algorithm: FindMax-CRCW

```
M ← vector of length n
parallel for 1 ≤ i ≤ n do
    M[i] ← 1
parallel for 1 ≤ i < j ≤ n do
    if X[i] < X[j] then M[i] ← 0;
parallel for 1 ≤ i ≤ n do
    if M[i] = 1 then result ← i;
return result
```

Number of processors used: $O(n^2)$
Time complexity:

Find max in CRCW PRAM

Suppose we have a vector X of length n. Our goal is to find the index of the maximum value in X.

A sequential algorithm has complexity $O(n)$.

Algorithm: FindMax-CRCW

$M \leftarrow$ vector of length n

parallel for $1 \leq i \leq n$ **do**

  $M[i] \leftarrow 1$

parallel for $1 \leq i < j \leq n$ **do**

  if $X[i] < X[j]$ **then** $M[i] \leftarrow 0$;

parallel for $1 \leq i \leq n$ **do**

  if $M[i] = 1$ **then** *result* $\leftarrow i$;

return *result*

Number of processors used: $O(n^2)$

Time complexity: $O(1)$

Work complexity:

Find max in CRCW PRAM

Suppose we have a vector X of length n. Our goal is to find the index of the maximum value in X.
A sequential algorithm has complexity $O(n)$.

Algorithm: FindMax-CRCW

$M \leftarrow$ vector of length n
parallel for $1 \leq i \leq n$ **do**
    $M[i] \leftarrow 1$
parallel for $1 \leq i < j \leq n$ **do**
    if $X[i] < X[j]$ **then** $M[i] \leftarrow 0$;
parallel for $1 \leq i \leq n$ **do**
    if $M[i] = 1$ **then** $result \leftarrow i$;
return $result$

Number of processors used: $O(n^2)$
Time complexity: $O(1)$
Work complexity: $O(n^2)$

Find max in CRCW PRAM

Suppose we have a vector X of length n. Our goal is to find the index of the maximum value in X.
A sequential algorithm has complexity $O(n)$.

Algorithm: FindMax-CRCW

$M \leftarrow$ vector of length n
parallel for $1 \leq i \leq n$ **do**
  $M[i] \leftarrow 1$
parallel for $1 \leq i < j \leq n$ **do**
  if $X[i] < X[j]$ **then** $M[i] \leftarrow 0$;
parallel for $1 \leq i \leq n$ **do**
  if $M[i] = 1$ **then** $result \leftarrow i$;
return *result*

Number of processors used: $O(n^2)$ – can be improved to $O(n^{1+\epsilon})$
Time complexity: $O(1)$
Work complexity: $O(n^2)$

Find first one in CRCW PRAM

Suppose we have a binary X of length n. Our goal is to find the index of the first 1 in X.

- Design an algorithm (for CRCW PRAM) with a constant time complexity, using $O(n^2)$ processors.

Find first one in CRCW PRAM

Suppose we have a binary X of length n. Our goal is to find the index of the first 1 in X.

- Design an algorithm (for CRCW PRAM) with a constant time complexity, using $O(n^2)$ processors.
- Design an algorithm (for CRCW PRAM) with a constant time complexity, using $O(n)$ processors.

EREW vs. CRCW algorithms

Finding 1 in a binary vector or finding a minimum in a vector of integers requires $O(\log n)$ time on EREW machine.
This is not an accident.

EREW vs. CRCW algorithms

Finding 1 in a binary vector or finding a minimum in a vector of integers requires $O(\log n)$ time on EREW machine.
This is not an accident.

Theorem

Every algorithm using p processors on CRCW machine can be at most $O(\log p)$ faster than the fastest algorithm for the same problem, using p processors on EREW machine.

EREW vs. CRCW algorithms

Finding 1 in a binary vector or finding a minimum in a vector of integers requires $O(\log n)$ time on EREW machine.
This is not an accident.

Theorem

Every algorithm using p processors on CRCW machine can be at most $O(\log p)$ faster than the fastest algorithm for the same problem, using p processors on EREW machine.

We can simulate any algorithm for CRCW PRAM on EREW PRAM with overhead at most $O(\log p)$.

EREW vs. CRCW algorithms

Finding 1 in a binary vector or finding a minimum in a vector of integers requires $O(\log n)$ time on EREW machine.
This is not an accident.

Theorem

Every algorithm using p processors on CRCW machine can be at most $O(\log p)$ faster than the fastest algorithm for the same problem, using p processors on EREW machine.

We can simulate any algorithm for CRCW PRAM on EREW PRAM with overhead at most $O(\log p)$.

From now on we focus on EREW PRAM.

Summation

Suppose we have a vector X of length n. Our goal is to find the sum of elements in X.

Summation

Suppose we have a vector X of length n. Our goal is to find the sum of elements in X. A sequential algorithm has complexity

Summation

Suppose we have a vector X of length n. Our goal is to find the sum of elements in X. A sequential algorithm has complexity $O(n)$.

Summation

Suppose we have a vector X of length n. Our goal is to find the sum of elements in X. A sequential algorithm has complexity $O(n)$. The following procedure computes the sum of $X[i, i+1, \ldots, j]$.

Algorithm: ParallelSum[X,i,j]

if $i = j$ **then return** $X[i]$;
$r \leftarrow [0, 0]$
parallel for $p = 1, 2$ **do**
  if $p = 1$ **then** $r[p] \leftarrow \mathit{ParallelSum}(X, i, \lfloor (i+j)/2 \rfloor)$;
  if $p = 2$ **then** $r[p] \leftarrow \mathit{ParallelSum}(X, \lfloor (i+j)/2 \rfloor + 1, j)$;
return $r[1] + r[2]$

Summation

Suppose we have a vector X of length n. Our goal is to find the sum of elements in X. A sequential algorithm has complexity $O(n)$. The following procedure computes the sum of $X[i, i+1, \ldots, j]$.

Algorithm: ParallelSum[X,i,j]

if $i = j$ **then return** $X[i]$;

$r \leftarrow [0, 0]$

parallel for $p = 1, 2$ **do**

  if $p = 1$ **then** $r[p] \leftarrow \mathit{ParallelSum}(X, i, \lfloor (i+j)/2 \rfloor)$;

  if $p = 2$ **then** $r[p] \leftarrow \mathit{ParallelSum}(X, \lfloor (i+j)/2 \rfloor + 1, j)$;

return $r[1] + r[2]$

Number of processors used:

Summation

Suppose we have a vector X of length n. Our goal is to find the sum of elements in X. A sequential algorithm has complexity $O(n)$. The following procedure computes the sum of $X[i, i+1, \ldots, j]$.

Algorithm: ParallelSum[X,i,j]

if $i = j$ **then return** $X[i]$;
$r \leftarrow [0, 0]$
parallel for $p = 1, 2$ **do**
  if $p = 1$ **then** $r[p] \leftarrow \mathit{ParallelSum}(X, i, \lfloor (i+j)/2 \rfloor)$;
  if $p = 2$ **then** $r[p] \leftarrow \mathit{ParallelSum}(X, \lfloor (i+j)/2 \rfloor + 1, j)$;
return $r[1] + r[2]$

Number of processors used: n

Time complexity:

Summation

Suppose we have a vector X of length n. Our goal is to find the sum of elements in X. A sequential algorithm has complexity $O(n)$. The following procedure computes the sum of $X[i, i+1, \dots, j]$.

Algorithm: ParallelSum[X,i,j]

if $i = j$ **then return** $X[i]$;
$r \leftarrow [0, 0]$
parallel for $p = 1, 2$ **do**
  if $p = 1$ **then** $r[p] \leftarrow ParallelSum(X, i, \lfloor (i+j)/2 \rfloor)$;
  if $p = 2$ **then** $r[p] \leftarrow ParallelSum(X, \lfloor (i+j)/2 \rfloor + 1, j)$;
return $r[1] + r[2]$

Number of processors used: n
Time complexity: $O(\log n)$
Work complexity:

Summation

Suppose we have a vector X of length n. Our goal is to find the sum of elements in X. A sequential algorithm has complexity $O(n)$. The following procedure computes the sum of $X[i, i+1, \ldots, j]$.

Algorithm: ParallelSum[X,i,j]

if $i = j$ **then return** $X[i]$;
$r \leftarrow [0, 0]$
parallel for $p = 1, 2$ **do**
  if $p = 1$ **then** $r[p] \leftarrow \mathit{ParallelSum}(X, i, \lfloor (i+j)/2 \rfloor)$;
  if $p = 2$ **then** $r[p] \leftarrow \mathit{ParallelSum}(X, \lfloor (i+j)/2 \rfloor + 1, j)$;
return $r[1] + r[2]$

Number of processors used: n
Time complexity: $O(\log n)$
Work complexity: $O(n \log n)$

Summation – non-recursive

Algorithm: ParallelSumNonRec[X]

```
parallel for p = 1, 2 do
    B[i] ← X[i]
for h = 1, 2, ..., log n do
    parallel for p = 1, 2, ..., n/2^h do
        B[i] ← B[2i − 1] + B[2i]
return B[1]
```

Summation – non-recursive

Algorithm: ParallelSumNonRec$[X]$

parallel for $p = 1, 2$ **do**
  $B[i] \leftarrow X[i]$

for $h = 1, 2, \ldots, \log n$ **do**
  parallel for $p = 1, 2, \ldots, n/2^h$ **do**
    $B[i] \leftarrow B[2i-1] + B[2i]$

return $B[1]$

Of course in the same way we can compute other functions, like maximum, minimum, boolean operations etc.

Mean square error – second approach

Algorithm: MSE2

$X2[\ldots] \leftarrow$ vector of length n

$E \leftarrow 0$

$sx \leftarrow 0$

$sx2 \leftarrow 0$

parallel foreach $i = 1, 2, \ldots, n$ **do**

  $X2[i] \leftarrow X[i] \cdot X[i]$

$sx \leftarrow ParallelSum(X, 1, n)$

$sx2 \leftarrow ParallelSum(X2, 1, n)$

$E \leftarrow sx2 - sx \cdot sx / n$

return E

Mean square error – second approach

Algorithm: MSE2

$X2[\ldots] \leftarrow$ vector of length n
$E \leftarrow 0$
$sx \leftarrow 0$
$sx2 \leftarrow 0$
parallel foreach $i = 1, 2, \ldots, n$ **do**
    $X2[i] \leftarrow X[i] \cdot X[i]$
$sx \leftarrow ParallelSum(X, 1, n)$
$sx2 \leftarrow ParallelSum(X2, 1, n)$
$E \leftarrow sx2 - sx \cdot sx/n$
return E

Number of processors used:

Mean square error – second approach

Algorithm: MSE2

$X2[\ldots] \leftarrow$ vector of length n
$E \leftarrow 0$
$sx \leftarrow 0$
$sx2 \leftarrow 0$
parallel foreach $i = 1, 2, \ldots, n$ **do**
    $X2[i] \leftarrow X[i] \cdot X[i]$
$sx \leftarrow ParallelSum(X, 1, n)$
$sx2 \leftarrow ParallelSum(X2, 1, n)$
$E \leftarrow sx2 - sx \cdot sx / n$
return E

Number of processors used: n

Time complexity:

Mean square error – second approach

Algorithm: MSE2

$X2[\ldots] \leftarrow$ vector of length n
$E \leftarrow 0$
$sx \leftarrow 0$
$sx2 \leftarrow 0$
parallel foreach $i = 1, 2, \ldots, n$ **do**
  $X2[i] \leftarrow X[i] \cdot X[i]$
$sx \leftarrow ParallelSum(X, 1, n)$
$sx2 \leftarrow ParallelSum(X2, 1, n)$
$E \leftarrow sx2 - sx \cdot sx/n$
return E

Number of processors used: n
Time complexity: $O(\log n)$
Work complexity:

Mean square error – second approach

Algorithm: MSE2

$X2[\ldots] \leftarrow$ vector of length n

$E \leftarrow 0$

$sx \leftarrow 0$

$sx2 \leftarrow 0$

parallel foreach $i = 1, 2, \ldots, n$ **do**

  $X2[i] \leftarrow X[i] \cdot X[i]$

$sx \leftarrow \mathit{ParallelSum}(X, 1, n)$

$sx2 \leftarrow \mathit{ParallelSum}(X2, 1, n)$

$E \leftarrow sx2 - sx \cdot sx/n$

return E

Number of processors used: n

Time complexity: $O(\log n)$

Work complexity: $O(n \log n)$

Prefix sum

Prefix sum

Prefix sum (sometimes called scan) is an operation that transforms a vector $X[x_1, x_2, \ldots, x_n]$ into a vector $[x'_1, x'_2, \ldots, x'_n]$, where $x'_i = \sum_{j=1}^{i} x_i$.
So for

$$X = [1, 1, 2, 0, 3, 1, 2],$$

we obtain

$$[1, 2, 4, 4, 7, 8, 10].$$

Of course wan change $+$ to any other binary operation, like max, min, multiplication, boolean function etc.

Prefix sum – example

Suppose we want to find all solutions of some problem. We have distributed the searching among n processors, each processor found some number of solutions (possible 0). It keeps them in its own set T. Our goal is to create table X, containing all solutions.

Algorithm: CollectSolutions

$num[\ldots] \leftarrow$ vector of length n
parallel foreach $i = 1, 2, \ldots, n$ **do**
    $num[i] \leftarrow |T|$
$Scan(num)$
$X \leftarrow$ vector of length $num[n]$
parallel foreach $i = 1, 2, \ldots, n$ **do**
    for $j = num[i-1] + 1$ **to** $num[i]$ **do**
        $X[j] \leftarrow T[j - num[i-1]]$
return X

To simplify the notation, we assume that $num[0] = 0$.

Parallel prefix sum – first approach

We can do something very similar to the parallel addition algorithm. Assume for simplicity that n is a power of 2.

Algorithm: Scan1(X)

if $|X| = 0$ **then return**;
parallel for $p = 1, 2$ **do**
    if $p = 1$ **then** $Scan1(LeftHalf(X))$;
    if $p = 2$ **then** $Scan1(RightHalf(X))$;
parallel for $n/2 < j \leq n$ **do**
    $X[j] \leftarrow x[n/2] + x[j]$

Parallel prefix sum – first approach

We can do something very similar to the parallel addition algorithm. Assume for simplicity that n is a power of 2.

Algorithm: Scan1(X)

if $|X| = 0$ **then return**;
parallel for $p = 1, 2$ **do**
  if $p = 1$ **then** $Scan1(LeftHalf(X))$;
  if $p = 2$ **then** $Scan1(RightHalf(X))$;
parallel for $n/2 < j \leq n$ **do**
  $X[j] \leftarrow x[n/2] + x[j]$

Number of processors used:

Parallel prefix sum – first approach

We can do something very similar to the parallel addition algorithm. Assume for simplicity that n is a power of 2.

Algorithm: Scan1(X)

if $|X| = 0$ **then return**;
parallel for $p = 1, 2$ **do**
    if $p = 1$ **then** $Scan1(LeftHalf(X))$;
    if $p = 2$ **then** $Scan1(RightHalf(X))$;
parallel for $n/2 < j \leq n$ **do**
    $X[j] \leftarrow x[n/2] + x[j]$

Number of processors used: n
Time complexity:

Parallel prefix sum – first approach

We can do something very similar to the parallel addition algorithm. Assume for simplicity that n is a power of 2.

Algorithm: Scan1(X)

```
if |X| = 0 then return;
parallel for p = 1, 2 do
    if p = 1 then  Scan1(LeftHalf(X)) ;
    if p = 2 then  Scan1(RightHalf(X)) ;
parallel for n/2 < j ≤ n do
    X[j] ← x[n/2] + x[j]
```

Number of processors used: n

Time complexity: $O(\log n)$ (exactly $\log n$ steps)

Work complexity:

Parallel prefix sum – first approach

We can do something very similar to the parallel addition algorithm. Assume for simplicity that n is a power of 2.

Algorithm: Scan1(X)

if $|X| = 0$ **then return**;
parallel for $p = 1, 2$ **do**
    if $p = 1$ **then** $Scan1(LeftHalf(X))$;
    if $p = 2$ **then** $Scan1(RightHalf(X))$;
parallel for $n/2 < j \leq n$ **do**
    $X[j] \leftarrow x[n/2] + x[j]$

Number of processors used: n
Time complexity: $O(\log n)$ (exactly $\log n$ steps)
Work complexity: $O(n \log n)$

Parallel prefix sum – second approach

Again, we assume that n is a power of 2.

Algorithm: Scan2(X)

if $|X| = 0$ **then return**;
$Y \leftarrow$ a vector of length $n/2$
parallel for $1 \leq i \leq n/2$ **do**
  $Y[i] \leftarrow X[2i-1] + X[2i]$
$Scan2(Y)$
parallel for $1 < i \leq n/2$ **do**
  $X[2i] \leftarrow Y[i]$
  if $i > 1$ **then** $X[2i-1] \leftarrow Y[i-1] + X[2i-1]$;

Parallel prefix sum – second approach

Again, we assume that n is a power of 2.

Algorithm: Scan2(X)

if $|X| = 0$ **then return**;
$Y \leftarrow$ a vector of length $n/2$
parallel for $1 \leq i \leq n/2$ **do**
    $Y[i] \leftarrow X[2i-1] + X[2i]$
$Scan2(Y)$
parallel for $1 < i \leq n/2$ **do**
    $X[2i] \leftarrow Y[i]$
    if $i > 1$ **then** $X[2i-1] \leftarrow Y[i-1] + X[2i-1]$;

Number of processors used:

Parallel prefix sum – second approach

Again, we assume that n is a power of 2.

Algorithm: Scan2(X)

if $|X| = 0$ **then return**;

$Y \leftarrow$ a vector of length $n/2$

parallel for $1 \leq i \leq n/2$ **do**

  $Y[i] \leftarrow X[2i-1] + X[2i]$

*Scan*2(Y)

parallel for $1 < i \leq n/2$ **do**

  $X[2i] \leftarrow Y[i]$

  if $i > 1$ **then** $X[2i-1] \leftarrow Y[i-1] + X[2i-1]$;

Number of processors used: can be efficiently made $O(n/\log n)$

Time complexity:

Parallel prefix sum – second approach

Again, we assume that n is a power of 2.

Algorithm: Scan2(X)

if $|X| = 0$ **then return**;
$Y \leftarrow$ a vector of length $n/2$
parallel for $1 \leq i \leq n/2$ **do**
    $Y[i] \leftarrow X[2i-1] + X[2i]$
Scan2(Y)
parallel for $1 < i \leq n/2$ **do**
    $X[2i] \leftarrow Y[i]$
    if $i > 1$ **then** $X[2i-1] \leftarrow Y[i-1] + X[2i-1]$;

Number of processors used: can be efficiently made $O(n/\log n)$
Time complexity: $O(\log n)$ (exactly $2\log n - 2$ steps)
Work complexity:

Parallel prefix sum – second approach

Again, we assume that n is a power of 2.

Algorithm: Scan2(X)

if $|X| = 0$ **then return**;
$Y \leftarrow$ a vector of length $n/2$
parallel for $1 \leq i \leq n/2$ **do**
  $Y[i] \leftarrow X[2i-1] + X[2i]$
Scan2(Y)
parallel for $1 < i \leq n/2$ **do**
  $X[2i] \leftarrow Y[i]$
  if $i > 1$ **then** $X[2i-1] \leftarrow Y[i-1] + X[2i-1]$;

Number of processors used: can be efficiently made $O(n/\log n)$
Time complexity: $O(\log n)$ (exactly $2\log n - 2$ steps)
Work complexity: $O(n)$

Parallel prefix sum – second approach

Again, we assume that n is a power of 2.

Algorithm: Scan2(X)

if $|X| = 0$ **then return**;
$Y \leftarrow$ a vector of length $n/2$
parallel for $1 \leq i \leq n/2$ **do**
    $Y[i] \leftarrow X[2i-1] + X[2i]$
$Scan2(Y)$
parallel for $1 < i \leq n/2$ **do**
    $X[2i] \leftarrow Y[i]$
    if $i > 1$ **then** $X[2i-1] \leftarrow Y[i-1] + X[2i-1]$;

Number of processors used: can be efficiently made $O(n/\log n)$
Time complexity: $O(\log n)$ (exactly $2\log n - 2$ steps)
Work complexity: $O(n)$
Works better if we have fewer processors

Matrix multiplication

Suppose we have to multiply two matrices A and B.

Matrix multiplication

Suppose we have to multiply two matrices A and B.
The simplest, sequential way to do this is:

Algorithm: Multiply(A, B)

```
for i = 1 to n do
    for j = 1 to n do
        c_{i,j} ← 0
        for k = 1 to n do
            c_{i,j} ← c_{i,j} + a_{i,k} · b_{k,j}
```

Matrix multiplication

Suppose we have to multiply two matrices A and B.
The simplest, sequential way to do this is:

Algorithm: Multiply(A, B)

for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
        $c_{i,j} \leftarrow 0$
        for $k = 1$ **to** n **do**
            $c_{i,j} \leftarrow c_{i,j} + a_{i,k} \cdot b_{k,j}$

How to do this in parallel? For simplicity, assume that:

- they are square matrices $n \times n$,
- the number of processors available is p^2,
- p divides n.

Parallel matrix multiplication – blocks

We partition our matrices into p^2 blocks.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,p} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,p} \\ & \vdots & & \\ A_{p,1} & A_{p,2} & \cdots & A_{p,p} \end{pmatrix}$$

$$B = \begin{pmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,p} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,p} \\ & \vdots & & \\ B_{p,1} & B_{p,2} & \cdots & B_{p,p} \end{pmatrix}$$

Parallel matrix multiplication – idea of the algorithm

Each process (i,j) computes a single block $C_{i,j}$. Each processes one block of A and one block of B at the time.

Algorithm: Compute(i,j)

$C_{i,j} \leftarrow p \times p$ matrix of zeros
for $k = 0$ **to** $p - 1$ **do**
    $C_{i,j} \leftarrow C_{i,j} + A_{i-k,j} \cdot B_{i,j-k}$

In each step we shift A to the left and shift B up (modulo p).
The multiplication inside the loop is just a sequential multiplication of two $\frac{n}{p} \times \frac{n}{p}$ matrices.

Parallel matrix multiplication – idea of the algorithm

Each process (i, j) computes a single block $C_{i,j}$. Each processes one block of A and one block of B at the time.

Algorithm: Compute(i,j)

$C_{i,j} \leftarrow p \times p$ matrix of zeros
for $k = 0$ **to** $p - 1$ **do**
  $C_{i,j} \leftarrow C_{i,j} + A_{i-k,j} \cdot B_{i,j-k}$

In each step we shift A to the left and shift B up (modulo p).

The multiplication inside the loop is just a sequential multiplication of two $\frac{n}{p} \times \frac{n}{p}$ matrices.

Time complexity:

Parallel matrix multiplication – idea of the algorithm

Each process (i, j) computes a single block $C_{i,j}$. Each processes one block of A and one block of B at the time.

Algorithm: Compute(i,j)

$C_{i,j} \leftarrow p \times p$ matrix of zeros
for $k = 0$ **to** $p - 1$ **do**
$\quad C_{i,j} \leftarrow C_{i,j} + A_{i-k,j} \cdot B_{i,j-k}$

In each step we shift A to the left and shift B up (modulo p).

The multiplication inside the loop is just a sequential multiplication of two $\frac{n}{p} \times \frac{n}{p}$ matrices.

Time complexity: $O\left(p \cdot (n/p)^3\right) = O\left(\frac{n^3}{p^2}\right)$ (if the naive algorithm for matrix multiplication is used).