Analysis of Algorithms

Multi-threaded Algorithms

Andres Mendez-Vazquez

April 15, 2016

Outline

Cinvestav

Outline

Multi-Threaded Algorithms

Motivation

- Until now, our serial algorithms are quite suitable for running on a single processor system.
- However, multiprocessor algorithms are ubiquitous:
    - Therefore, extending our serial models to a parallel computation model is a must.

    - Shared Memory
    - Message Passing
    - Etc.

Multi-Threaded Algorithms

Motivation

- Until now, our serial algorithms are quite suitable for running on a single processor system.
- However, multiprocessor algorithms are ubiquitous:
  - Therefore, extending our serial models to a parallel computation model is a must.

Computational Model

- There exist many competing models of parallel computation that are essentially different:
  - Shared Memory
  - Message Passing
  - Etc.

Cinvestav

Multi-Threaded Algorithms

Motivation

- Until now, our serial algorithms are quite suitable for running on a single processor system.
- However, multiprocessor algorithms are ubiquitous:
    - Therefore, extending our serial models to a parallel computation model is a must.

Computational Model

- There exist many competing models of parallel computation that are essentially different:
    - Shared Memory
    - Message Passing
    - Etc.

Multi-Threaded Algorithms

Motivation

- Until now, our serial algorithms are quite suitable for running on a single processor system.
- However, multiprocessor algorithms are ubiquitous:
  - Therefore, extending our serial models to a parallel computation model is a must.

Computational Model

- There exist many competing models of parallel computation that are essentially different:
  - Shared Memory
  - Message Passing
  - Etc.

Multi-Threaded Algorithms

Motivation

- Until now, our serial algorithms are quite suitable for running on a single processor system.
- However, multiprocessor algorithms are ubiquitous:
  - Therefore, extending our serial models to a parallel computation model is a must.

Computational Model

- There exist many competing models of parallel computation that are essentially different:
  - Shared Memory
  - Message Passing
  - Etc.

Multi-Threaded Algorithms

Motivation

- Until now, our serial algorithms are quite suitable for running on a single processor system.
- However, multiprocessor algorithms are ubiquitous:
  - Therefore, extending our serial models to a parallel computation model is a must.

Computational Model

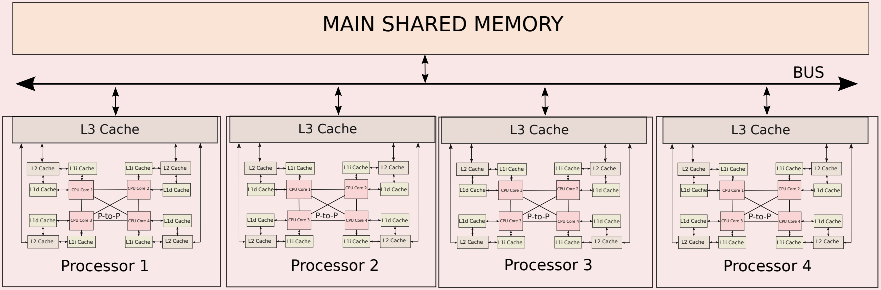
- There exist many competing models of parallel computation that are essentially different:
  - Shared Memory
  - Message Passing
  - Etc.

Outline

The Model to Be Used

Symmetric Multiprocessor

The model that we will use is the Symmetric Multiprocessor (SMP) where a shared memory exists.



Dynamic Multi-Threading

Dynamic Multi-Threading

- In reality it can be difficult to handle multi-threaded programs in a SMP.
- Thus, we will assume a simple concurrency platform that handles all the resources:
    - Schedules
    - Memory
    - Etc
- It is Called **Dynamic Multi-threading**.

- Sync
- Parallel

Dynamic Multi-Threading

Dynamic Multi-Threading

- In reality it can be difficult to handle multi-threaded programs in a SMP.
- Thus, we will assume a simple concurrency platform that handles all the resources:
  - Schedules
  - Memory
  - Etc
- It is Called **Dynamic Multi-threading**.

Dynamic Multi-Threading Computing Operations

- Spawn
- Sync
- Parallel

Dynamic Multi-Threading

Dynamic Multi-Threading

- In reality it can be difficult to handle multi-threaded programs in a SMP.
- Thus, we will assume a simple concurrency platform that handles all the resources:
  - Schedules
  - Memory
  - Etc
- It is Called **Dynamic Multi-threading**.

Dynamic Multi-Threading Computing Operations

- Spawn
- Sync
- Parallel

Dynamic Multi-Threading

Dynamic Multi-Threading

- In reality it can be difficult to handle multi-threaded programs in a SMP.
- Thus, we will assume a simple concurrency platform that handles all the resources:
  - Schedules
  - Memory
  - Etc
- It is Called **Dynamic Multi-threading**.

Dynamic Multi-Threading Computing Operations

- Spawn
- Sync
- Parallel

Dynamic Multi-Threading

Dynamic Multi-Threading

- In reality it can be difficult to handle multi-threaded programs in a SMP.
- Thus, we will assume a simple concurrency platform that handles all the resources:
  - Schedules
  - Memory
  - Etc
- It is Called **Dynamic Multi-threading**.

Dynamic Multi-Threading Computing Operations

- Spawn
- Sync
- Parallel

Dynamic Multi-Threading

Dynamic Multi-Threading

- In reality it can be difficult to handle multi-threaded programs in a SMP.
- Thus, we will assume a simple concurrency platform that handles all the resources:
    - Schedules
    - Memory
    - Etc
- It is Called **Dynamic Multi-threading**.

Dynamic Multi-Threading Computing Operations

- Spawn
- Sync
- Parallel

Dynamic Multi-Threading

Dynamic Multi-Threading

- In reality it can be difficult to handle multi-threaded programs in a SMP.
- Thus, we will assume a simple concurrency platform that handles all the resources:
    - Schedules
    - Memory
    - Etc
- It is Called **Dynamic Multi-threading**.

Dynamic Multi-Threading Computing Operations

- Spawn
- Sync
- Parallel

Dynamic Multi-Threading

Dynamic Multi-Threading

- In reality it can be difficult to handle multi-threaded programs in a SMP.
- Thus, we will assume a simple concurrency platform that handles all the resources:
  - Schedules
  - Memory
  - Etc
- It is Called **Dynamic Multi-threading**.

Dynamic Multi-Threading Computing Operations

- Spawn
- Sync
- Parallel

Dynamic Multi-Threading

Dynamic Multi-Threading

- In reality it can be difficult to handle multi-threaded programs in a SMP.
- Thus, we will assume a simple concurrency platform that handles all the resources:
  - Schedules
  - Memory
  - Etc
- It is Called **Dynamic Multi-threading**.

Dynamic Multi-Threading Computing Operations

- Spawn
- Sync
- Parallel

Dynamic Multi-Threading

Dynamic Multi-Threading

- In reality it can be difficult to handle multi-threaded programs in a SMP.
- Thus, we will assume a simple concurrency platform that handles all the resources:
  - Schedules
  - Memory
  - Etc
- It is Called **Dynamic Multi-threading**.

Dynamic Multi-Threading Computing Operations

- Spawn
- Sync
- Parallel

Outline

SPAWN

SPAWN

When called before a procedure, the parent procedure may continue to execute in parallel.

- The Scheduler decide which computations should run concurrently

SPAWN

SPAWN

When called before a procedure, the parent procedure may continue to execute in parallel.

Note

- The keyword **spawn** does not say anything about concurrent execution, but it can happen.
- The Scheduler decide which computations should run concurrently.

SPAWN

SPAWN

When called before a procedure, the parent procedure may continue to execute in parallel.

Note

- The keyword **spawn** does not say anything about concurrent execution, but it can happen.
- The Scheduler decide which computations should run concurrently.

SYNC AND PARALLEL

SYNC

The keyword sync indicates that the procedure must wait for all its spawned children to complete.

PARALLEL

This operation applies to loops, which make possible to execute the body of the loop in parallel.

SYNC AND PARALLEL

SYNC

The keyword sync indicates that the procedure must wait for all its spawned children to complete.

PARALLEL

This operation applies to loops, which make possible to execute the body of the loop in parallel.

Outline

1. Introduction
    - Why Multi-Threaded Algorithms?
2. Model To Be Used
    - Symmetric Multiprocessor
    - Operations
    - Example
3. Computation DAG
    - Introduction
4. Performance Measures
    - Introduction
    - Running Time Classification
5. Parallel Laws
    - Work and Span Laws
    - Speedup and Parallelism
    - Greedy Scheduler
    - Scheduling Rises the Following Issue
6. Examples
    - Parallel Fibonacci
    - Matrix Multiplication
    - Parallel Merge-Sort
7. Exercises
    - Some Exercises you can try!!!

A Classic Parallel Piece of Code: Fibonacci Numbers

Fibonacci's Definition

- $F_0 = 0$
- $F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

Naive Algorithm

```
$Fibonacci(n)$
if $n \leq 1$ then
        return $n$
else $x = Fibonacci(n-1)$
        $y = Fibonacci(n-2)$
        return $x + y$
```

A Classic Parallel Piece of Code: Fibonacci Numbers

Fibonacci's Definition

- $F_0 = 0$
- $F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

Naive Algorithm

$Fibonacci(n)$

```
if $n \leq 1$ then
        return $n$
else $x = Fibonacci(n-1)$
        $y = Fibonacci(n-2)$
        return $x + y$
```

Time Complexity

Recursion and Complexity

- Recursion $T(n) = T(n-1) + T(n-2) + \Theta(1)$.
- Complexity $T(n) = \Theta(F_n) = \Theta(\phi^n)$, $\phi = \frac{1+\sqrt{5}}{2}$.

Time Complexity

Recursion and Complexity

- Recursion $T\left(n\right)=T\left(n-1\right)+T\left(n-2\right)+\Theta\left(1\right)$.
- Complexity $T(n)=\Theta\left(F_n\right)=\Theta\left(\phi^n\right)$, $\phi=\frac{1+\sqrt{5}}{2}$.

There is a Better Way

We can order the first tree numbers in the sequence as

$$\begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Then

$$\begin{aligned} \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \\ &= \begin{pmatrix} F_3 & F_2 \\ F_2 & F_1 \end{pmatrix} \end{aligned}$$

There is a Better Way

We can order the first tree numbers in the sequence as

$$\begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Then

$$\begin{aligned} \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \\ &= \begin{pmatrix} F_3 & F_2 \\ F_2 & F_1 \end{pmatrix} \end{aligned}$$

There is a Better Way

Calculating in $O(\log n)$ when n is a power of 2

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n} = \begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix}$$

Thus

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\frac{n}{2}} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\frac{n}{2}} = \begin{pmatrix} F\left(\frac{n}{2}+1\right) & F\left(\frac{n}{2}\right) \\ F\left(\frac{n}{2}\right) & F\left(\frac{n}{2}-1\right) \end{pmatrix} \begin{pmatrix} F\left(\frac{n}{2}+1\right) & F\left(\frac{n}{2}\right) \\ F\left(\frac{n}{2}\right) & F\left(\frac{n}{2}-1\right) \end{pmatrix}$$

However...

We will use the naive version to illustrate the principles of parallel programming.

There is a Better Way

Calculating in $O(\log n)$ when n is a power of 2

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n} = \begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix}$$

Thus

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\frac{n}{2}} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\frac{n}{2}} = \begin{pmatrix} F\left(\frac{n}{2}+1\right) & F\left(\frac{n}{2}\right) \\ F\left(\frac{n}{2}\right) & F\left(\frac{n}{2}-1\right) \end{pmatrix} \begin{pmatrix} F\left(\frac{n}{2}+1\right) & F\left(\frac{n}{2}\right) \\ F\left(\frac{n}{2}\right) & F\left(\frac{n}{2}-1\right) \end{pmatrix}$$

However...

We will use the naive version to illustrate the principles of parallel programming.

The Concurrent Code

Parallel Algorithm

$PFibonacci(n)$

```
if $n \leq 1$ then
        return $n$
else $x =$ spawn $Fibonacci(n-1)$
        $y = Fibonacci(n-2)$
        sync
        return $x + y$
```

The Concurrent Code

Parallel Algorithm

PFibonacci(n)

```
if $n \leq 1$ then
        return $n$
else $x =$ **spawn** $Fibonacci(n-1)$
        $y = Fibonacci(n-2)$
        sync
        return $x + y$
```

The Concurrent Code

Parallel Algorithm

```
$PFibonacci(n)$
if $n \leq 1$ then
        return $n$
else $x =$ **spawn** $Fibonacci(n-1)$
        $y = Fibonacci(n-2)$
        **sync**
        return $x + y$
```

Outline

How do we compute a complexity? Computation DAG

Definition

A directed acyclic $G = (V, E)$ graph where

- The vertices V are sets of instructions.
- The edges E represent dependencies between sets of instructions i.e. (u, v) instruction u before v.

How do we compute a complexity? Computation DAG

Definition

A directed acyclic $G = (V, E)$ graph where

- The vertices V are sets of instructions.
- The edges E represent dependencies between sets of instructions i.e. (u, v) instruction u before v.

Notes

- A set of instructions without **any** parallel control are grouped in a **strand**.
- Thus, V represents a set of strands and E represents dependencies between strands induced by parallel control.
- A strand of maximal length will be called a **thread**.

How do we compute a complexity? Computation DAG

Definition

A directed acyclic $G = (V, E)$ graph where

- The vertices V are sets of instructions.
- The edges E represent dependencies between sets of instructions i.e. (u, v) instruction u before v.

Notes

- A set of instructions without **any** parallel control are grouped in a **strand**.
- Thus, V represents a set of strands and E represents dependencies between strands induced by parallel control.
- A strand of maximal length will be called a **thread**.

How do we compute a complexity? Computation DAG

Definition

A directed acyclic $G = (V, E)$ graph where

- The vertices V are sets of instructions.
- The edges E represent dependencies between sets of instructions i.e. (u, v) instruction u before v.

Notes

- A set of instructions without **any** parallel control are grouped in a **strand**.
- Thus, V represents a set of strands and E represents dependencies between strands induced by parallel control.
- A strand of maximal length will be called a **thread**.

How do we compute a complexity? Computation DAG

Definition

A directed acyclic $G = (V, E)$ graph where

- The vertices V are sets of instructions.
- The edges E represent dependencies between sets of instructions i.e. (u, v) instruction u before v.

Notes

- A set of instructions without **any** parallel control are grouped in a **strand**.
- Thus, V represents a set of strands and E represents dependencies between strands induced by parallel control.
- A strand of maximal length will be called a **thread**.

How do we compute a complexity? Computation DAG

Definition

A directed acyclic $G = (V, E)$ graph where

- The vertices V are sets of instructions.
- The edges E represent dependencies between sets of instructions i.e. (u, v) instruction u before v.

Notes

- A set of instructions without **any** parallel control are grouped in a **strand**.
- Thus, V represents a set of strands and E represents dependencies between strands induced by parallel control.
- A strand of maximal length will be called a **thread**.

How do we compute a complexity? Computation DAG

Thus

- If there is an edge between thread u and v, then they are said to be (logically) in series.
- If there is no edge, then they are said to be (logically) in parallel.

Cinvestav

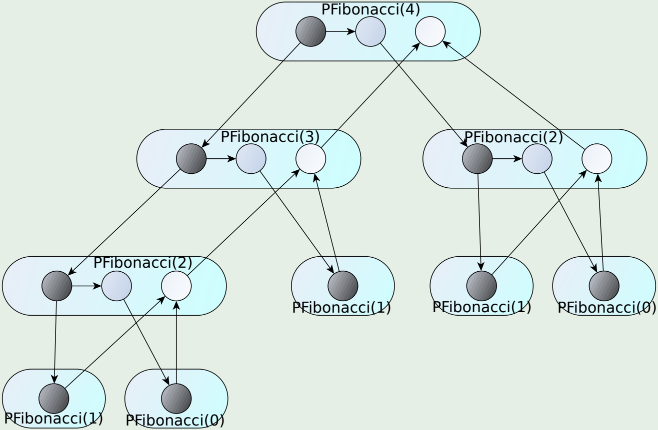
How do we compute a complexity? Computation DAG

Thus

- If there is an edge between thread u and v, then they are said to be (logically) in series.
- If there is no edge, then they are said to be (logically) in parallel.

Example: PFibonacci(4)

Example



Edge Classification

Continuation Edge

A **continuation edge** (u, v) connects a thread u to its successor v within the same procedure instance.

Spawned Edge

When a thread u spawns a new thread v, then (u, v) is called a **spawned edge.**

Call Edges

Call edges represent normal procedure call.

Return Edge

Return edge signals when a thread v returns to its calling procedure.

Edge Classification

Continuation Edge

A **continuation edge** (u, v) connects a thread u to its successor v within the same procedure instance.

Spawned Edge

When a thread u spawns a new thread v, then (u, v) is called a **spawned edge.**

Call Edges

Call edges represent normal procedure call.

Return Edge

Return edge signals when a thread v returns to its calling procedure.

Edge Classification

Continuation Edge

A **continuation edge** (u, v) connects a thread u to its successor v within the same procedure instance.

Spawned Edge

When a thread u spawns a new thread v, then (u, v) is called a **spawned edge.**

Call Edges

Call edges represent normal procedure call.

Return Edge

Return edge signals when a thread v returns to its calling procedure.

Edge Classification

Continuation Edge

A **continuation edge** (u, v) connects a thread u to its successor v within the same procedure instance.

Spawned Edge

When a thread u spawns a new thread v, then (u, v) is called a **spawned edge.**

Call Edges

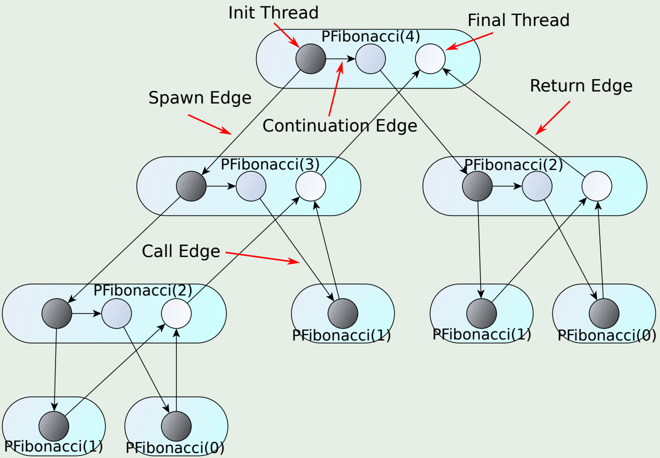
Call edges represent normal procedure call.

Return Edge

Return edge signals when a thread v returns to its calling procedure.

Example: PFibonacci(4)

The Different Edges



Outline

Performance Measures

WORK

The work of a multi-threaded computation is the total time to execute the entire computation on **one processor.**

$$Work = \sum_{i \in I} Time\left(Thread_i\right)$$

SPAN

The span is the longest time to execute the strands along any path of the DAG.

Performance Measures

WORK

The work of a multi-threaded computation is the total time to execute the entire computation on **one processor.**

$$Work = \sum_{i \in I} Time\left(Thread_i\right)$$

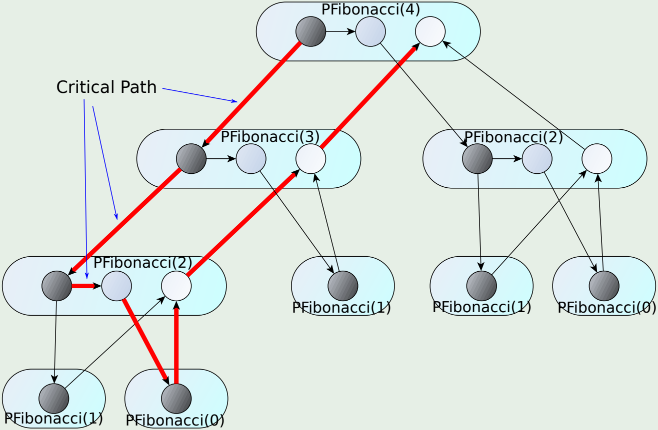
SPAN

The span is the longest time to execute the strands along any path of the DAG.

- In a DAG which each strand takes unit time, the span equals the number of vertices on a longest or **critical path** in the DAG.

Example: PFibonacci(4)

Example



Example

Example

- In Fibonacci(4), we have
    - 17 threads.
    - 8 vertices in the longest path

Example

Example

- In Fibonacci(4), we have
    - 17 threads.
    - 8 vertices in the longest path

We have that

- Assuming unit time
    - WORK=17 time units
    - SPAN=8 time units

    - Available Cores
    - Scheduler Policies

Example

Example

- In Fibonacci(4), we have
  - 17 threads.
  - 8 vertices in the longest path

We have that

- Assuming unit time
  - WORK=17 time units
  - SPAN=8 time units

Note

- Running time not only depends on work and span but
  - Available Cores
  - Scheduler Policies

Example

Example

- In Fibonacci(4), we have
  - 17 threads.
  - 8 vertices in the longest path

We have that

- Assuming unit time
  - WORK=17 time units
  - SPAN=8 time units

Note

- Running time not only depends on work and span but
  - Available Cores
  - Scheduler Policies

Example

Example

- In Fibonacci(4), we have
    - 17 threads.
    - 8 vertices in the longest path

We have that

- Assuming unit time
    - WORK=17 time units
    - SPAN=8 time units

Note

- Running time not only depends on work and span but
    - Available Cores
    - Scheduler Policies

Example

Example

- In Fibonacci(4), we have
  - 17 threads.
  - 8 vertices in the longest path

We have that

- Assuming unit time
  - WORK=17 time units
  - SPAN=8 time units

Note

- Running time not only depends on work and span but
  - Available Cores
  - Scheduler Policies

Example

Example

- In Fibonacci(4), we have
    - 17 threads.
    - 8 vertices in the longest path

We have that

- Assuming unit time
    - WORK=17 time units
    - SPAN=8 time units

Note

- Running time not only depends on work and span but
    - Available Cores
    - Scheduler Policies

Example

Example

- In Fibonacci(4), we have
    - 17 threads.
    - 8 vertices in the longest path

We have that

- Assuming unit time
    - WORK=17 time units
    - SPAN=8 time units

Note

- Running time not only depends on work and span but
    - Available Cores
    - Scheduler Policies

Example

Example

- In Fibonacci(4), we have
  - 17 threads.
  - 8 vertices in the longest path

We have that

- Assuming unit time
  - WORK=17 time units
  - SPAN=8 time units

Note

- Running time not only depends on work and span but
  - Available Cores
  - Scheduler Policies

Outline

Running Time Classification

Single Processor

- T_1 running time on a single processor.

Multiple Processors

- T_p running time on P processors.

Unlimited Processors

- T_∞ running time on unlimited processors, also called the span, if we run each strand on its own processor.

Running Time Classification

Single Processor

- T_1 running time on a single processor.

Multiple Processors

- T_p running time on P processors.

Unlimited Processors

- T_∞ running time on unlimited processors, also called the span, if we run each strand on its own processor.

Running Time Classification

Single Processor

- T_1 running time on a single processor.

Multiple Processors

- T_p running time on P processors.

Unlimited Processors

- T_∞ running time on unlimited processors, also called the span, if we run each strand on its own processor.

Outline

Work Law

Definition

- In one step, an ideal parallel computer with P processors can do:
  - At most P units of work.
  - Thus in T_P time, it can perform at most PT_P work.

$$PT_P \geq T_1 \Longrightarrow T_p \geq \frac{T_1}{P}$$

Work Law

Definition

- In one step, an ideal parallel computer with P processors can do:
  - At most P units of work.
  - Thus in T_P time, it can perform at most PT_P work.

$$PT_P \geq T_1 \Longrightarrow T_p \geq \frac{T_1}{P}$$

Cinvestav

Work Law

Definition

- In one step, an ideal parallel computer with P processors can do:
  - At most P units of work.
  - Thus in T_P time, it can perform at most PT_P work.

$$PT_P \geq T_1 \Longrightarrow T_p \geq \frac{T_1}{P}$$

Work Law

Definition

- In one step, an ideal parallel computer with P processors can do:
  - At most P units of work.
  - Thus in T_P time, it can perform at most PT_P work.

$$PT_P \geq T_1 \Longrightarrow T_p \geq \frac{T_1}{P}$$

Span Law

Definition

- A P-processor ideal parallel computer cannot run faster than a machine with unlimited number of processors.
- However, a computer with unlimited number of processors can emulate a P-processor machine by using simply P of its processors. Therefore,

$$T_P \geq T_\infty$$

Span Law

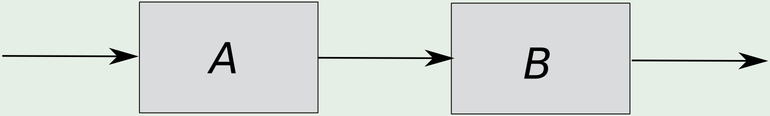
Definition

- A P-processor ideal parallel computer cannot run faster than a machine with unlimited number of processors.
- However, a computer with unlimited number of processors can emulate a P-processor machine by using simply P of its processors. Therefore,

$$T_P \geq T_\infty$$

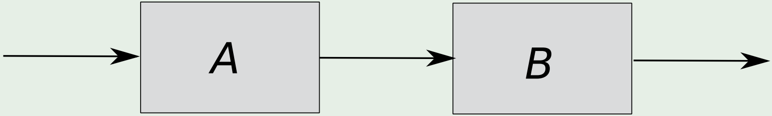
Work Calculations: Serial

Serial Computations



Work Calculations: Serial

Serial Computations

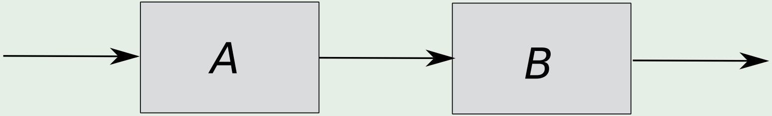


Note

- Work: $T_1 \left(A \cup B \right) = T_1 \left(A \right) + T_1 \left(B \right)$.
- Span: $T_\infty \left(A \cup B \right) = T_\infty \left(A \right) + T_\infty \left(B \right)$.

Work Calculations: Serial

Serial Computations

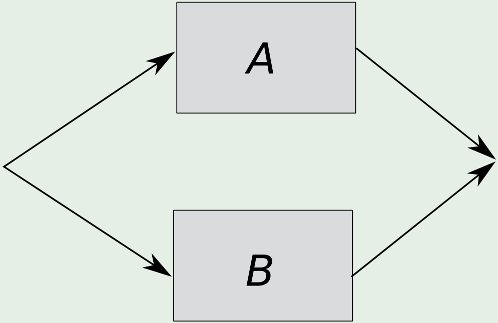


Note

- Work: $T_1 (A \cup B) = T_1 (A) + T_1 (B)$.
- Span: $T_\infty (A \cup B) = T_\infty (A) + T_\infty (B)$.

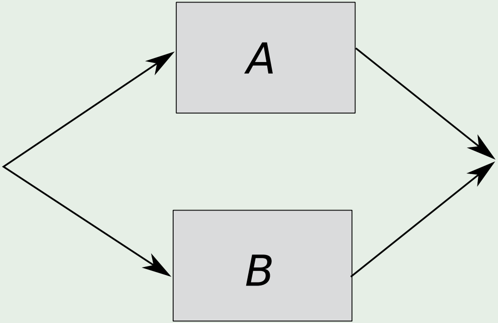
Work Calculations: Parallel

Parallel Computations



Work Calculations: Parallel

Parallel Computations

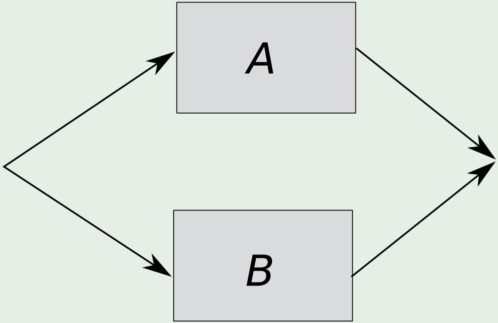


Note

- Work: $T_1 (A \cup B) = T_1 (A) + T_1 (B)$.
- Span: $T_\infty (A \cup B) = \max \{T_\infty (A), T_\infty (B)\}$.

Work Calculations: Parallel

Parallel Computations



Note

- Work: $T_1(A \cup B) = T_1(A) + T_1(B)$.
- Span: $T_\infty(A \cup B) = \max\{T_\infty(A), T_\infty(B)\}$.

Outline

Speedup and Parallelism

Speed up

- The speed up of a computation on P processors is defined as $\frac{T_1}{T_P}$.
- Then, by work law $\frac{T_1}{T_P} \leq P$. Thus, the speedup on P processors can be at most P.

Speedup and Parallelism

Speed up

- The speed up of a computation on P processors is defined as $\frac{T_1}{T_P}$.
- Then, by work law $\frac{T_1}{T_P} \leq P$. Thus, the speedup on P processors can be at most P.

Notes

- Linear Speedup when $\frac{T_1}{T_P} = \Theta(P)$.
- Perfect Linear Speedup when $\frac{T_1}{T_P} = P$.

  - In specific, we are looking to have a lot of parallelism
  - This changes from Algorithm to Algorithm

Speedup and Parallelism

Speed up

- The speed up of a computation on P processors is defined as $\frac{T_1}{T_P}$.
- Then, by work law $\frac{T_1}{T_P} \leq P$. Thus, the speedup on P processors can be at most P.

Notes

- Linear Speedup when $\frac{T_1}{T_P} = \Theta\left(P\right)$.
- Perfect Linear Speedup when $\frac{T_1}{T_P} = P$.

Parallelism

- The parallelism of a computation on P processors is defined as $\frac{T_1}{T_\infty}$.
  - In specific, we are looking to have a lot of parallelism.
  - This changes from Algorithm to Algorithm.

Speedup and Parallelism

Speed up

- The speed up of a computation on P processors is defined as $\frac{T_1}{T_P}$.
- Then, by work law $\frac{T_1}{T_P} \leq P$. Thus, the speedup on P processors can be at most P.

Notes

- Linear Speedup when $\frac{T_1}{T_P} = \Theta\left(P\right)$.
- Perfect Linear Speedup when $\frac{T_1}{T_P} = P$.

Parallelism

- The parallelism of a computation on P processors is defined as $\frac{T_1}{T_\infty}$.
    - In specific, we are looking to have a lot of parallelism.
    - This changes from Algorithm to Algorithm.

Speedup and Parallelism

Speed up

- The speed up of a computation on P processors is defined as $\frac{T_1}{T_P}$.
- Then, by work law $\frac{T_1}{T_P} \leq P$. Thus, the speedup on P processors can be at most P.

Notes

- Linear Speedup when $\frac{T_1}{T_P} = \Theta\left(P\right)$.
- Perfect Linear Speedup when $\frac{T_1}{T_P} = P$.

Parallelism

- The parallelism of a computation on P processors is defined as $\frac{T_1}{T_\infty}$.
    - In specific, we are looking to have a lot of parallelism.
    - This changes from Algorithm to Algorithm.

Speedup and Parallelism

Speed up

- The speed up of a computation on P processors is defined as $\frac{T_1}{T_P}$.
- Then, by work law $\frac{T_1}{T_P} \leq P$. Thus, the speedup on P processors can be at most P.

Notes

- Linear Speedup when $\frac{T_1}{T_P} = \Theta(P)$.
- Perfect Linear Speedup when $\frac{T_1}{T_P} = P$.

Parallelism

- The parallelism of a computation on P processors is defined as $\frac{T_1}{T_\infty}$.
  - In specific, we are looking to have a lot of parallelism.
  - This changes from Algorithm to Algorithm.

Speedup and Parallelism

Speed up

- The speed up of a computation on P processors is defined as $\frac{T_1}{T_P}$.
- Then, by work law $\frac{T_1}{T_P} \leq P$. Thus, the speedup on P processors can be at most P.

Notes

- Linear Speedup when $\frac{T_1}{T_P} = \Theta(P)$.
- Perfect Linear Speedup when $\frac{T_1}{T_P} = P$.

Parallelism

- The parallelism of a computation on P processors is defined as $\frac{T_1}{T_\infty}$.
  - In specific, we are looking to have a lot of parallelism.
  - This changes from Algorithm to Algorithm.

Outline

Greedy Scheduler

Definition

- A **greedy scheduler** assigns as many strands to processors as possible in each time step.

Greedy Scheduler

Definition

- A **greedy scheduler** assigns as many strands to processors as possible in each time step.

Note

- On P processors, if at least P strands are ready to execute during a time step, then we say that the step is a **complete step.**
- Otherwise we say that it is an **incomplete step**.
- This changes from Algorithm to Algorithm.

Greedy Scheduler

Definition

- A **greedy scheduler** assigns as many strands to processors as possible in each time step.

Note

- On P processors, if at least P strands are ready to execute during a time step, then we say that the step is a **complete step.**
- Otherwise we say that it is an **incomplete step**.
- This changes from Algorithm to Algorithm.

Greedy Scheduler

Definition

- A **greedy scheduler** assigns as many strands to processors as possible in each time step.

Note

- On P processors, if at least P strands are ready to execute during a time step, then we say that the step is a **complete step.**
- Otherwise we say that it is an **incomplete step**.
- This changes from Algorithm to Algorithm.

Greedy Scheduler Theorem and Corollaries

Theorem 27.1

On an ideal parallel computer with P processors, a greedy scheduler executes a multi-threaded computation with work T_1 and span T_∞ in time $T_P \leq \frac{T_1}{P} + T_\infty$.

Corollary 27.2

The running time T_P of any multi-threaded computation scheduled by a greedy scheduler on an ideal parallel computer with P processors is within a factor of 2 of optimal.

Corollary 27.3

Let T_P be the running time of a multi-threaded computation produced by a greedy scheduler on an ideal parallel computer with P processors, and let T_1 and T_∞ be the work and span of the computation, respectively. Then, if $P \ll \frac{T_1}{T_\infty}$ (Much Less), we have $T_P \approx \frac{T_1}{P}$, or equivalently, a speedup of approximately P .

Greedy Scheduler Theorem and Corollaries

Theorem 27.1

On an ideal parallel computer with P processors, a greedy scheduler executes a multi-threaded computation with work T_1 and span T_∞ in time $T_P \leq \frac{T_1}{P} + T_\infty$.

Corollary 27.2

The running time T_P of any multi-threaded computation scheduled by a greedy scheduler on an ideal parallel computer with P processors is within a factor of 2 of optimal.

Corollary 27.3

Let T_P be the running time of a multi-threaded computation produced by a greedy scheduler on an ideal parallel computer with P processors, and let T_1 and T_∞ be the work and span of the computation, respectively. Then, if $P \ll \frac{T_1}{T_\infty}$ (Much Less), we have $T_P \approx \frac{T_1}{P}$, or equivalently, a speedup of approximately P.

Greedy Scheduler Theorem and Corollaries

Theorem 27.1

On an ideal parallel computer with P processors, a greedy scheduler executes a multi-threaded computation with work T_1 and span T_∞ in time $T_P \leq \frac{T_1}{P} + T_\infty$.

Corollary 27.2

The running time T_P of any multi-threaded computation scheduled by a greedy scheduler on an ideal parallel computer with P processors is within a factor of 2 of optimal.

Corollary 27.3

Let T_P be the running time of a multi-threaded computation produced by a greedy scheduler on an ideal parallel computer with P processors, and let T_1 and T_∞ be the work and span of the computation, respectively. Then, if $P \ll \frac{T_1}{T_\infty}$ (Much Less), we have $T_P \approx \frac{T_1}{P}$, or equivalently, a speedup of approximately P .

Outline

Race Conditions

Determinacy Race

A determinacy race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

Example

Race-Example()

```
$x = 0$
parallel for $i = 1$ to 3 do
    $x = x + 1$
print $x$
```

Race Conditions

Determinacy Race

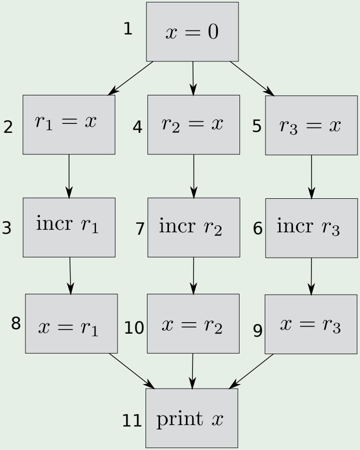
A determinacy race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

Example

```
Race-Example()
    $x = 0$
    parallel for $i = 1$ to 3 do
        $x = x + 1$
    print $x$
```

Example

Determinacy Race Example



step	x	r_1	r_2	r_3
1	0			
2	0	0		
3	0	1		
4	0	1	0	
5	0	1	0	0
6	0	1	0	1
7	0	1	1	1
8	1	1	1	1
9	1	1	1	1
10	1	1	1	1

Cinvestav

Example

NOTE

Although, this is of great importance is beyond the scope of this class:

- For More about this topic, we have:
  - Maurice Herlihy and Nir Shavit, "*The Art of Multiprocessor Programming*," Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
  - Andrew S. Tanenbaum, "*Modern Operating Systems*" (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

Example

NOTE

Although, this is of great importance is beyond the scope of this class:

- For More about this topic, we have:
  - Maurice Herlihy and Nir Shavit, "*The Art of Multiprocessor Programming*," Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
  - Andrew S. Tanenbaum, "*Modern Operating Systems*" (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

Example

NOTE

Although, this is of great importance is beyond the scope of this class:

- For More about this topic, we have:
  - Maurice Herlihy and Nir Shavit, "*The Art of Multiprocessor Programming*," Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
  - Andrew S. Tanenbaum, "*Modern Operating Systems*" (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

Example

NOTE

Although, this is of great importance is beyond the scope of this class:

- For More about this topic, we have:
    - Maurice Herlihy and Nir Shavit, "*The Art of Multiprocessor Programming*," Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
    - Andrew S. Tanenbaum, "*Modern Operating Systems*" (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

Outline

Example of Complexity: PFibonacci

Complexity

$$T_\infty(n) = \max\left\{T_\infty(n-1), T_\infty(n-2)\right\} + \Theta(1)$$

Finally

$$T_\infty(n) = T_\infty(n-1) + \Theta(1) = \Theta(n)$$

Parallelism

$$\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{\phi^n}{n}\right)$$

Example of Complexity: PFibonacci

Complexity

$$T_\infty(n) = \max\left\{T_\infty(n-1), T_\infty(n-2)\right\} + \Theta(1)$$

Finally

$$T_\infty(n) = T_\infty(n-1) + \Theta(1) = \Theta(n)$$

Parallelism

$$\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{\phi^n}{n}\right)$$

Cinvestav

Example of Complexity: PFibonacci

Complexity

$$T_{\infty}(n) = \max\left\{T_{\infty}(n-1), T_{\infty}(n-2)\right\} + \Theta(1)$$

Finally

$$T_{\infty}(n) = T_{\infty}(n-1) + \Theta(1) = \Theta(n)$$

Parallelism

$$\frac{T_1(n)}{T_{\infty}(n)} = \Theta\left(\frac{\phi^n}{n}\right)$$

Outline

1. Introduction
    - Why Multi-Threaded Algorithms?
2. Model To Be Used
    - Symmetric Multiprocessor
    - Operations
    - Example
3. Computation DAG
    - Introduction
4. Performance Measures
    - Introduction
    - Running Time Classification
5. Parallel Laws
    - Work and Span Laws
    - Speedup and Parallelism
    - Greedy Scheduler
    - Scheduling Rises the Following Issue
6. **Examples**
    - Parallel Fibonacci
    - **Matrix Multiplication**
    - Parallel Merge-Sort
7. Exercises
    - Some Exercises you can try!!!

Matrix Multiplication

Trick

To multiply two $n \times n$ matrices, we perform 8 matrix multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and one addition $n \times n$ of matrices.

Idea

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \dots$$

$$\begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$$

Matrix Multiplication

Trick

To multiply two $n \times n$ matrices, we perform 8 matrix multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and one addition $n \times n$ of matrices.

Idea

$$A=\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B=\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C=\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$C=\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}=\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}=\ldots$$

$$\begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix}+\begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$$

Any Idea to Parallelize the Code?

What do you think?

Did you notice the multiplications of sub-matrices?

Then What?

We have for example $A_{11}B_{11}$ and $A_{12}B_{21}$!!!

We can do the following

$$A_{11}B_{11} + A_{12}B_{21}$$

Any Idea to Parallelize the Code?

What do you think?

Did you notice the multiplications of sub-matrices?

Then What?

We have for example $A_{11}B_{11}$ and $A_{12}B_{21}$!!!

We can do the following

$$A_{11}B_{11} + A_{12}B_{21}$$

Any Idea to Parallelize the Code?

What do you think?

Did you notice the multiplications of sub-matrices?

Then What?

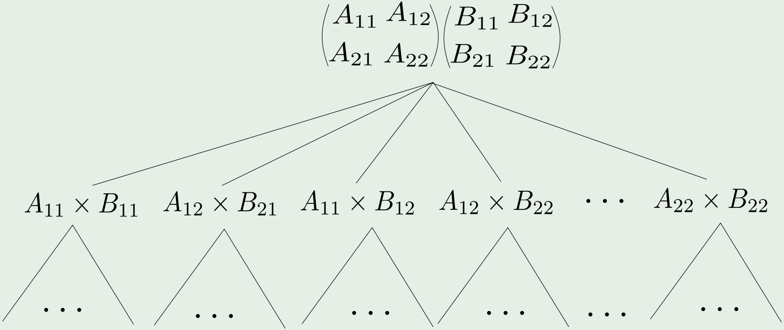
We have for example $A_{11}B_{11}$ and $A_{12}B_{21}$!!!

We can do the following

$$A_{11}B_{11} + A_{12}B_{21}$$

The use of the recursion!!!

As always our friend!!!



$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$A_{11} \times B_{11}$ $A_{12} \times B_{21}$ $A_{11} \times B_{12}$ $A_{12} \times B_{22}$ $\cdots$ $A_{22} \times B_{22}$

$\dots$ $\dots$ $\dots$ $\dots$ $\dots$ $\dots$

Pseudo-code of Matrix-Multiply

$Matrix-Multiply(C, A, B, n)$ **// The result of $A \times B$ in C with n a power of 2 for simplicity**

```
if $(n == 1)$
        $C[1,1] = A[1,1] + B[1,1]$
else
        allocate a temporary matrix $T[1 \ldots n, 1 \ldots n]$
        partition $A, B, C, T$ into $\frac{n}{2} \times \frac{n}{2}$ sub-matrices
        spawn $Matrix-Multiply(C_{11}, A_{11}, B_{11}, n/2)$
        spawn $Matrix-Multiply(C_{12}, A_{11}, B_{12}, n/2)$
        spawn $Matrix-Multiply(C_{21}, A_{21}, B_{11}, n/2)$
        spawn $Matrix-Multiply(C_{22}, A_{21}, B_{12}, n/2)$
        spawn $Matrix-Multiply(T_{11}, A_{12}, B_{21}, n/2)$
        spawn $Matrix-Multiply(T_{12}, A_{12}, B_{21}, n/2)$
        spawn $Matrix-Multiply(T_{21}, A_{22}, B_{21}, n/2)$
        $Matrix-Multiply(T_{22}, A_{22}, B_{22}, n/2)$
        sync
        $Matrix-Add(C, T, n)$
```

Pseudo-code of Matrix-Multiply

$Matrix-Multiply(C, A, B, n)$ **// The result of $A \times B$ in C with n a power of 2 for simplicity**

```
if $(n == 1)$
        $C[1,1] = A[1,1] + B[1,1]$
else
        allocate a temporary matrix $T[1...n, 1...n]$
        partition $A, B, C, T$ into $\frac{n}{2} \times \frac{n}{2}$ sub-matrices
        spawn $Matrix-Multiply(C_{11}, A_{11}, B_{11}, n/2)$
        spawn $Matrix-Multiply(C_{12}, A_{11}, B_{12}, n/2)$
        spawn $Matrix-Multiply(C_{21}, A_{21}, B_{11}, n/2)$
        spawn $Matrix-Multiply(C_{22}, A_{21}, B_{12}, n/2)$
        spawn $Matrix-Multiply(T_{11}, A_{12}, B_{21}, n/2)$
        spawn $Matrix-Multiply(T_{12}, A_{12}, B_{21}, n/2)$
        spawn $Matrix-Multiply(T_{21}, A_{22}, B_{21}, n/2)$
        $Matrix-Multiply(T_{22}, A_{22}, B_{22}, n/2)$
        sync
        $Matrix-Add(C, T, n)$
```

Pseudo-code of Matrix-Multiply

$Matrix - Multiply(C, A, B, n)$ // The result of $A \times B$ in C with n a power of 2 for simplicity

```
if $(n == 1)$
        $C[1,1] = A[1,1] + B[1,1]$
else
        allocate a temporary matrix $T[1...n, 1...n]$
        partition $A, B, C, T$ into $\frac{n}{2} \times \frac{n}{2}$ sub-matrices
        spawn $Matrix - Multiply(C_{11}, A_{11}, B_{11}, n/2)$
        spawn $Matrix - Multiply(C_{12}, A_{11}, B_{12}, n/2)$
        spawn $Matrix - Multiply(C_{21}, A_{21}, B_{11}, n/2)$
        spawn $Matrix - Multiply(C_{22}, A_{21}, B_{12}, n/2)$
        spawn $Matrix - Multiply(T_{11}, A_{12}, B_{21}, n/2)$
        spawn $Matrix - Multiply(T_{12}, A_{12}, B_{21}, n/2)$
        spawn $Matrix - Multiply(T_{21}, A_{22}, B_{21}, n/2)$
        $Matrix - Multiply(T_{22}, A_{22}, B_{22}, n/2)$
        sync
        $Matrix - Add(C, T, n)$
```

Pseudo-code of Matrix-Multiply

$Matrix-Multiply(C, A, B, n)$ **// The result of $A \times B$ in C with n a power of 2 for simplicity**

```
if $(n == 1)$
    $C[1,1] = A[1,1] + B[1,1]$
else
    allocate a temporary matrix $T[1...n, 1...n]$
    partition $A, B, C, T$ into $\frac{n}{2} \times \frac{n}{2}$ sub-matrices
    spawn $Matrix-Multiply(C_{11}, A_{11}, B_{11}, n/2)$
    spawn $Matrix-Multiply(C_{12}, A_{11}, B_{12}, n/2)$
    spawn $Matrix-Multiply(C_{21}, A_{21}, B_{11}, n/2)$
    spawn $Matrix-Multiply(C_{22}, A_{21}, B_{12}, n/2)$
    spawn $Matrix-Multiply(T_{11}, A_{12}, B_{21}, n/2)$
    spawn $Matrix-Multiply(T_{12}, A_{12}, B_{21}, n/2)$
    spawn $Matrix-Multiply(T_{21}, A_{22}, B_{21}, n/2)$
    $Matrix-Multiply(T_{22}, A_{22}, B_{22}, n/2)$
    sync
    $Matrix-Add(C, T, n)$
```

Pseudo-code of Matrix-Multiply

```
$Matrix - Multiply(C, A, B, n)$ // The result of $A \times B$ in $C$ with $n$ a power of 2 for simplicity
if $(n == 1)$
    $C[1,1] = A[1,1] + B[1,1]$
else
    allocate a temporary matrix $T[1 \ldots n, 1 \ldots n]$
    partition $A, B, C, T$ into $\frac{n}{2} \times \frac{n}{2}$ sub-matrices
    spawn $Matrix - Multiply(C_{11}, A_{11}, B_{11}, n/2)$
    spawn $Matrix - Multiply(C_{12}, A_{11}, B_{12}, n/2)$
    spawn $Matrix - Multiply(C_{21}, A_{21}, B_{11}, n/2)$
    spawn $Matrix - Multiply(C_{22}, A_{21}, B_{12}, n/2)$
    spawn $Matrix - Multiply(T_{11}, A_{12}, B_{21}, n/2)$
    spawn $Matrix - Multiply(T_{12}, A_{12}, B_{21}, n/2)$
    spawn $Matrix - Multiply(T_{21}, A_{22}, B_{21}, n/2)$
    $Matrix - Multiply(T_{22}, A_{22}, B_{22}, n/2)$
    sync
    $Matrix - Add(C, T, n)$
```

Explanation

Lines 1 - 2

Stops the recursion once you have only two numbers to multiply

Line 4

Extra matrix for storing the second matrix in

$$\begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \underbrace{\begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}}_{T}$$

Line 5

Do the desired partition!!!

Explanation

Lines 1 - 2

Stops the recursion once you have only two numbers to multiply

Line 4

Extra matrix for storing the second matrix in

$$\begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \underbrace{\begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}}_{T}$$

Line 5

Do the desired partition!!!

Explanation

Lines 1 - 2

Stops the recursion once you have only two numbers to multiply

Line 4

Extra matrix for storing the second matrix in

$$\begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \underbrace{\begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}}_{T}$$

Line 5

Do the desired partition!!!

Explanation

Lines 6 to 13

Calculating the products in

$$\begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$$

Using Recursion and Parallel Computations

Line 14

A barrier to wait until all the parallel computations are done!!!

Line 15

Call $Matrix - Add$ to add C and T.

Explanation

Lines 6 to 13

Calculating the products in

$$\begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$$

Using Recursion and Parallel Computations

Line 14

A barrier to wait until all the parallel computations are done!!!

Line 15

Call $Matrix - Add$ to add C and T.

Explanation

Lines 6 to 13

Calculating the products in

$$\begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$$

Using Recursion and Parallel Computations

Line 14

A barrier to wait until all the parallel computations are done!!!

Line 15

Call $Matrix - Add$ to add C and T.

Matrix ADD

Matrix Add Code

```
$Matrix - Add(C, T, n)$
// Add matrices $C$ and $T$ in-place to produce $C = C + T$
if $(n == 1)$
        $C[1,1] = C[1,1] + T[1,1]$
else
        Partition $C$ and $T$ into $\frac{n}{2} \times \frac{n}{2}$ sub-matrices
        spawn $Matrix - Add(C_{11}, T_{11}, n/2)$
        spawn $Matrix - Add(C_{12}, T_{12}, n/2)$
        spawn $Matrix - Add(C_{21}, T_{21}, n/2)$
        $Matrix - Add(C_{22}, T_{22}, n/2)$
        sync
```

Matrix ADD

Matrix Add Code

```
$Matrix - Add(C, T, n)$
// Add matrices $C$ and $T$ in-place to produce $C = C + T$
if $(n == 1)$
        $C[1,1] = C[1,1] + T[1,1]$
else
        Partition $C$ and $T$ into $\frac{n}{2} \times \frac{n}{2}$ sub-matrices
        spawn $Matrix - Add(C_{11}, T_{11}, n/2)$
        spawn $Matrix - Add(C_{12}, T_{12}, n/2)$
        spawn $Matrix - Add(C_{21}, T_{21}, n/2)$
        $Matrix - Add(C_{22}, T_{22}, n/2)$
        sync
```

Matrix ADD

Matrix Add Code

```
$Matrix - Add(C, T, n)$
// Add matrices $C$ and $T$ in-place to produce $C = C + T$
if $(n == 1)$
        $C[1,1] = C[1,1] + T[1,1]$
else
        Partition $C$ and $T$ into $\frac{n}{2} \times \frac{n}{2}$ sub-matrices
        spawn $Matrix - Add\left(C_{11}, T_{11}, {}^{n}/2\right)$
        spawn $Matrix - Add\left(C_{12}, T_{12}, {}^{n}/2\right)$
        spawn $Matrix - Add\left(C_{21}, T_{21}, {}^{n}/2\right)$
        $Matrix - Add\left(C_{22}, T_{22}, {}^{n}/2\right)$
        sync
```

Matrix ADD

Matrix Add Code

```
$Matrix - Add(C, T, n)$
// Add matrices $C$ and $T$ in-place to produce $C = C + T$
if $(n == 1)$
    $C[1,1] = C[1,1] + T[1,1]$
else
    Partition $C$ and $T$ into $\frac{n}{2} \times \frac{n}{2}$ sub-matrices
    spawn $Matrix - Add(C_{11}, T_{11}, {}^{n}/{2})$
    spawn $Matrix - Add(C_{12}, T_{12}, {}^{n}/{2})$
    spawn $Matrix - Add(C_{21}, T_{21}, {}^{n}/{2})$
    $Matrix - Add(C_{22}, T_{22}, {}^{n}/{2})$
    sync
```

Explanation

Line 1 - 2

Stops the recursion once you have only two numbers to multiply

Explanation

Line 1 - 2

Stops the recursion once you have only two numbers to multiply

Line 4

To Partition

- $C = \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix}$
- $T = \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$

Explanation

Line 1 - 2

Stops the recursion once you have only two numbers to multiply

Line 4

To Partition

- $C = \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix}$
- $T = \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$

In lines 5 to 8

We do the following sum in parallel!!!

$$\underbrace{\begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix}}_{C} + \underbrace{\begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}}_{T}$$

Explanation

Line 1 - 2

Stops the recursion once you have only two numbers to multiply

Line 4

To Partition

- $C = \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix}$
- $T = \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$

In lines 5 to 8

We do the following sum in parallel!!!

$$\underbrace{\begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix}}_{C} + \underbrace{\begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}}_{T}$$

Calculating Complexity of Matrix Multiplication

Work of Matrix Multiplication

The work of $T_1(n)$ of matrix multiplication satisfies the recurrence:

$$T_1(n) = \underbrace{8\,T_1\left(\frac{n}{2}\right)}_{\text{The sequential product}} + \underbrace{\Theta\left(n^2\right)}_{\text{The sequential sum}} = \Theta\left(n^3\right).$$

Calculating Complexity of Matrix Multiplication

Span of Matrix Multiplication

$$T_{\infty}(n) = \underbrace{T_{\infty}\left(\frac{n}{2}\right)}_{\text{The parallel product}} + \underbrace{\Theta(\log n)}_{\text{The parallel sum}} = \Theta\left(\log^2 n\right)$$

This is because:

Calculating Complexity of Matrix Multiplication

Span of Matrix Multiplication

$$T_{\infty}(n) = \underbrace{T_{\infty}\left(\frac{n}{2}\right)}_{\text{The parallel product}} + \underbrace{\Theta(\log n)}_{\text{The parallel sum}} = \Theta\left(\log^2 n\right)$$

This is because:

- $T_{\infty}\left(\frac{n}{2}\right)$ Matrix Multiplication is taking $\frac{n}{2} \times \frac{n}{2}$ matrices at the same time because parallelism.
- $\Theta(\log n)$ is the span of the addition of the matrices (Remember, we are using unlimited processors) which has a critical path of length $\log n$.

Calculating Complexity of Matrix Multiplication

Span of Matrix Multiplication

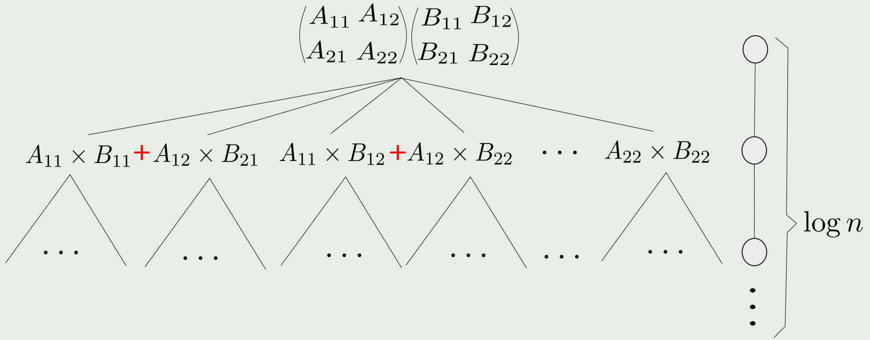
$$T_\infty(n) = \underbrace{T_\infty\left(\frac{n}{2}\right)}_{\text{The parallel product}} + \underbrace{\Theta(\log n)}_{\text{The parallel sum}} = \Theta\left(\log^2 n\right)$$

This is because:

- $T_\infty\left(\frac{n}{2}\right)$ Matrix Multiplication is taking $\frac{n}{2} \times \frac{n}{2}$ matrices at the same time because parallelism.
- $\Theta(\log n)$ is the span of the addition of the matrices (Remember, we are using unlimited processors) which has a critical path of length $\log n$.

Collapsing the sum

Parallel Sum



How much Parallelism?

The Final Parallelism in this Algorithm is

$$\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{n^3}{\log^2 n}\right)$$

Quite A Lot!!!

Outline

Merge-Sort : The Serial Version

We have

$Merge - Sort\,(A, p, r)$

Observation: **Sort elements in** $A\,[p...r]$

```
if $(p < r)$ then
    $q = \left\lfloor (p+r)/2 \right\rfloor$
    $Merge - Sort\,(A, p, q)$
    $Merge - Sort\,(A, q+1, r)$
    $Merge\,(A, p, q, r)$
```

Merge-Sort : The Parallel Version

We have

$Merge-Sort\left(A,p,r\right)$

Observation: **Sort elements in** $A\left[p...r\right]$

```
1. if $(p < r)$ then
2.         $q = \lfloor ^{(p+r)}/_{2} \rfloor$
3.         spawn $Merge-Sort\left(A,p,q\right)$
4.         $Merge-Sort\left(A,q+1,r\right)$ // Not necessary to spawn this
5.         sync
6.         $Merge\left(A,p,q,r\right)$
```

Calculating Complexity of This simple Parallel Merge-Sort

Work of Merge-Sort

- The work of $T_1(n)$ of this Parallel Merge-Sort satisfies the recurrence:

$$T_1(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T_1\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases} = \Theta(n \log n)$$

Because the Master Theorem Case 2.

Calculating Complexity of This simple Parallel Merge-Sort

Work of Merge-Sort

- The work of $T_1(n)$ of this Parallel Merge-Sort satisfies the recurrence:

$$T_1(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T_1\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases} = \Theta(n \log n)$$

Because the Master Theorem Case 2.

Span

$$T_\infty(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T_\infty\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases}$$

We have then

- $T_\infty\left(\frac{n}{2}\right)$ sort is taking two sorts at the same time because parallelism.
- **Then, $T_\infty(n) = \Theta(n)$ because the Master Theorem Case 3.**

Calculating Complexity of This simple Parallel Merge-Sort

Work of Merge-Sort

- The work of $T_1(n)$ of this Parallel Merge-Sort satisfies the recurrence:

$$T_1(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T_1\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases} = \Theta(n \log n)$$

  Because the Master Theorem Case 2.

Span

$$T_\infty(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T_\infty\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases}$$

We have then

- $T_\infty\left(\frac{n}{2}\right)$ sort is taking two sorts at the same time because parallelism.
- Then, $T_\infty(n) = \Theta(n)$ because the Master Theorem Case 3.

Calculating Complexity of This simple Parallel Merge-Sort

Work of Merge-Sort

- The work of $T_1(n)$ of this Parallel Merge-Sort satisfies the recurrence:

$$T_1(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T_1\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases} = \Theta(n \log n)$$

Because the Master Theorem Case 2.

Span

$$T_\infty(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T_\infty\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases}$$

We have then

- $T_\infty\left(\frac{n}{2}\right)$ sort is taking two sorts at the same time because parallelism.
- **Then,** $T_\infty(n) = \Theta(n)$ **because the Master Theorem Case 3.**

How much Parallelism?

The Final Parallelism in this Algorithm is

$$\frac{T_1(n)}{T_\infty(n)} = \Theta(\log n)$$

NOT NOT A Lot!!!

Can we improve this?

We have a problem

We have a bottleneck!!! Where?

Yes in the Merge part!!!

We need to improve that bottleneck!!!

Can we improve this?

We have a problem

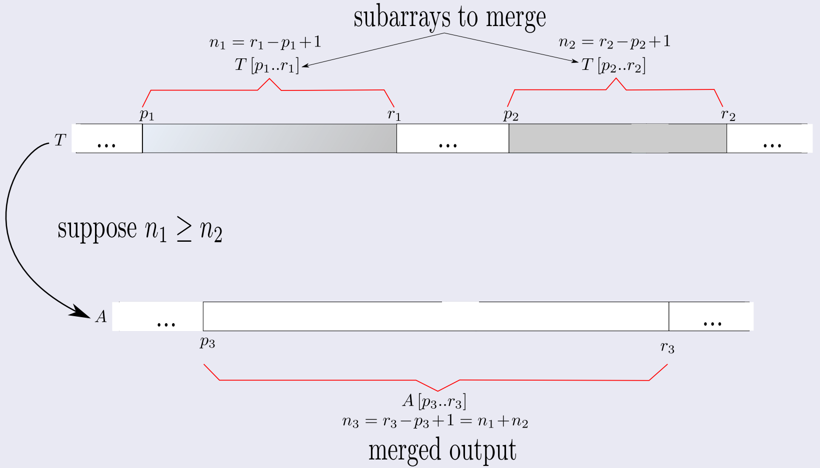
We have a bottleneck!!! Where?

Yes in the Merge part!!!

We need to improve that bottleneck!!!

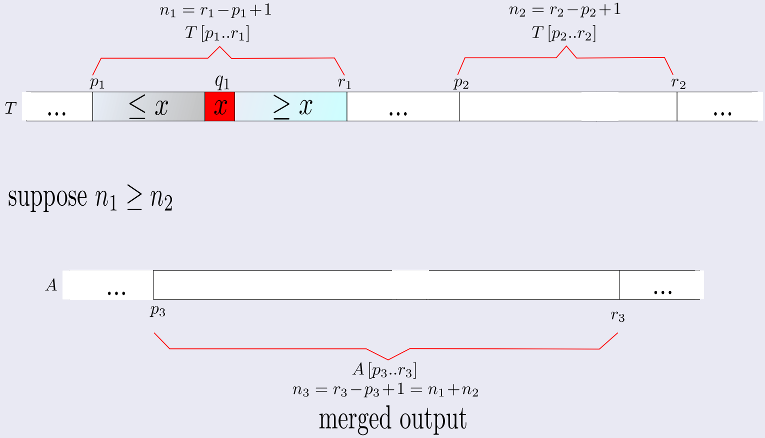
Parallel Merge

Example: Here, we use and intermediate array T



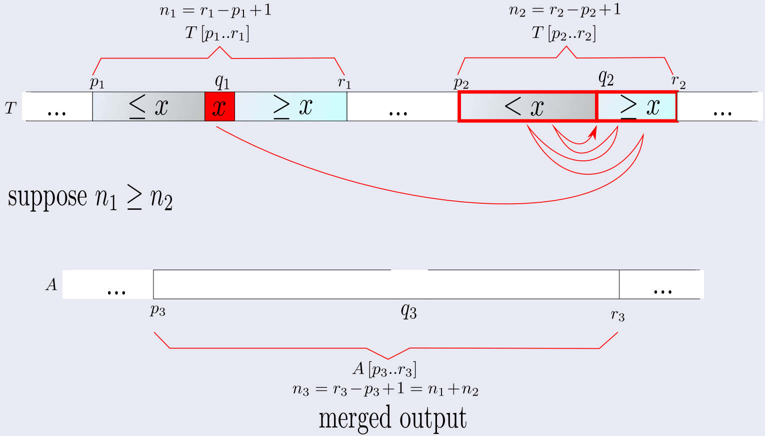
Parallel Merge

Step 1. Find $x = T[q_1]$ where $q_1 = \lfloor (p_1+r_1)/2 \rfloor$ or the midpoint in $T[p_1..r_1]$



Parallel Merge

Step 2. Use Binary Search in $T[p_1..r_1]$ to find q_2



Then

So that if we insert x between $T[q_2 - 1]$ and $T[q_2]$

$T\left[\begin{array}{ccccccc} p_1 & \cdots & q_2-1 & x & q_2 & \cdots & r_1 \end{array}\right]$ is sorted

Binary Search

It takes a key x and a sub-array $T[p..r]$ and it does

1. If $T[p..r]$ is empty $r < p$, then it returns the index p.
2. if $x \leq T[p]$, then it returns p.
3. if $x > T[p]$, then it returns the largest index q in the range $p < q \leq r + 1$ such that $T[q-1] < x$.

Binary Search

It takes a key x and a sub-array $T[p..r]$ and it does

1. If $T[p..r]$ is empty $r < p$, then it returns the index p.
2. if $x \leq T[p]$, then it returns p.
3. if $x > T[p]$, then it returns the largest index q in the range $p < q \leq r + 1$ such that $T[q - 1] < x$.

Binary Search

It takes a key x and a sub-array $T[p..r]$ and it does

1. If $T[p..r]$ is empty $r < p$, then it returns the index p.
2. if $x \leq T[p]$, then it returns p.
3. if $x > T[p]$, then it returns the largest index q in the range $p < q \leq r + 1$ such that $T[q-1] < x$.

Binary Search Code

```
BINARY-SEARCH($x$, $T$, $p$, $r$)
$low = p$
$high = \max\{p, r+1\}$
while $low < high$
    $mid = \left\lfloor \frac{log+high}{2} \right\rfloor$
    if $x \leq T[mid]$
        $high = mid$
    else $low = mid + 1$
return $high$
```

Binary Search Code

```
BINARY-SEARCH($x$, $T$, $p$, $r$)
$low = p$
$high = \max\{p, r+1\}$
while $low < high$
        $mid = \left\lfloor \frac{log+high}{2} \right\rfloor$
        if $x \leq T[mid]$
                $high = mid$
        else $low = mid + 1$
return $high$
```

Binary Search Code

```
BINARY-SEARCH($x, T, p, r$)
low = p
high = max{p, r+1}
while low < high
    $mid = \left\lfloor \frac{log+high}{2} \right\rfloor$
    if $x \leq T[mid]$
        high = mid
    else low = mid + 1
return high
```

Binary Search Code

BINARY-SEARCH(x, T, p, r)

```
$low = p$
$high = \max\{p, r+1\}$
while $low < high$
    $mid = \left\lfloor \frac{log + high}{2} \right\rfloor$
    if $x \leq T[mid]$
        $high = mid$
    else $low = mid + 1$
return $high$
```

Cinvestav

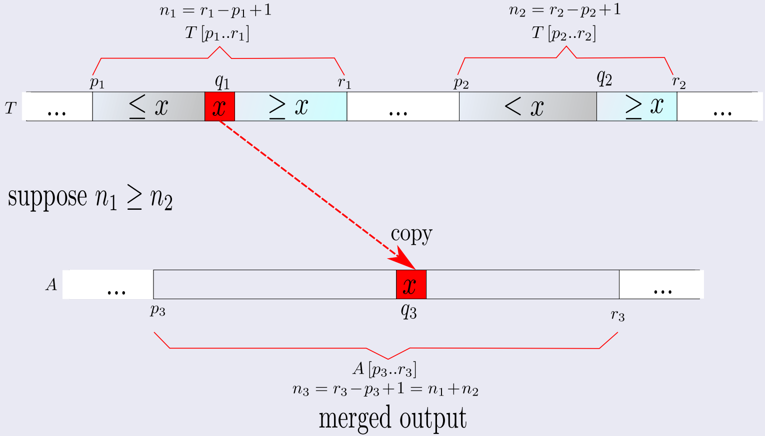
Binary Search Code

BINARY-SEARCH(x, T, p, r)

```
$low = p$
$high = \max\{p, r + 1\}$
while $low < high$
        $mid = \left\lfloor \frac{log + high}{2} \right\rfloor$
        if $x \leq T[mid]$
                $high = mid$
        else $low = mid + 1$
return $high$
```

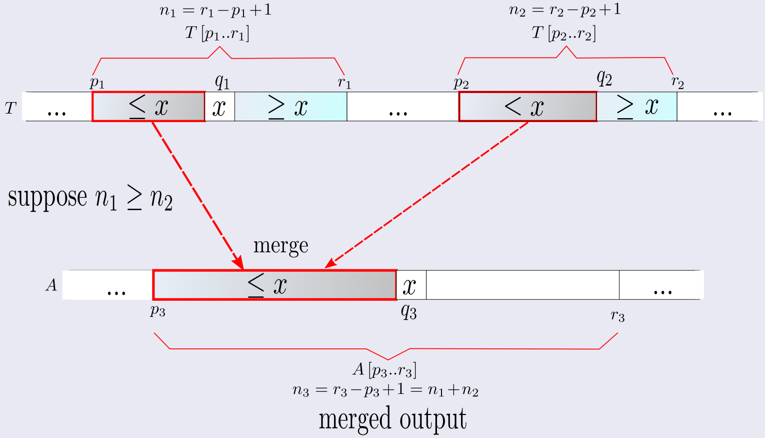
Parallel Merge

Step 3. Copy x in $A[q_3]$ where $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$



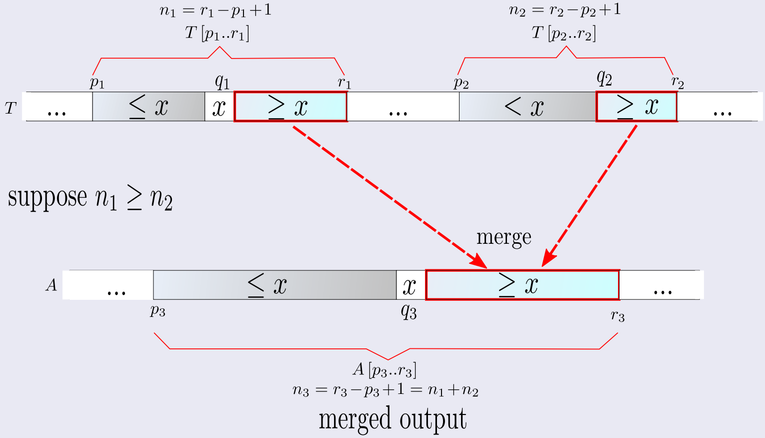
Parallel Merge

Step 4. Recursively merge $T[p_1..q_1-1]$ and $T[p_2..q_2-1]$ and place result into $A[p_3..q_3-1]$



Parallel Merge

Step 5. Recursively merge $T[q_1+1..r_1]$ and $T[q_2..r_2]$ and place result into $A[q_3+1..r_3]$



The Final Code for Parallel Merge

```
$Par - Merge\left(T, p_1, r_1, p_2, r_2, A, p_3\right)$
    $n_1 = r_1 - p_1 + 1$, $n_2 = r_2 - p_2 + 1$
    if $n_1 < n_2$
        Exchange $p_1 \leftrightarrow p_2, r_1 \leftrightarrow r_2, n_1 \leftrightarrow n_2$
    if $(n_1 == 0)$
        return
    else
        $q_1 = \lfloor (p_1+r_1)/2 \rfloor$
        $q_2 = BinarySearch\left(T\left[q_1\right], T, p_2, r_2\right)$
        $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
        $A\left[q_3\right] = T\left[q_1\right]$
        spawn $Par - Merge\left(T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3\right)$
        $Par - Merge\left(T, q_1 + 1, r_1, q_2 + 1, r_2, A, q_3 + 1\right)$
        sync
```

The Final Code for Parallel Merge

```
$Par - Merge\,(T, p_1, r_1, p_2, r_2, A, p_3)$
    $n_1 = r_1 - p_1 + 1$, $n_2 = r_2 - p_2 + 1$
    if $n_1 < n_2$
        Exchange $p_1 \leftrightarrow p_2, r_1 \leftrightarrow r_2, n_1 \leftrightarrow n_2$
    if $(n_1 == 0)$
        return
    else
        $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
        $q_2 = BinarySearch\,(T[q_1], T, p_2, r_2)$
        $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
        $A[q_3] = T[q_1]$
        spawn $Par - Merge\,(T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3)$
        $Par - Merge\,(T, q_1 + 1, r_1, q_2 + 1, r_2, A, q_3 + 1)$
        sync
```

The Final Code for Parallel Merge

```
$Par-Merge(T, p_1, r_1, p_2, r_2, A, p_3)$
$n_1 = r_1 - p_1 + 1$, $n_2 = r_2 - p_2 + 1$
if $n_1 < n_2$
        Exchange $p_1 \leftrightarrow p_2, r_1 \leftrightarrow r_2, n_1 \leftrightarrow n_2$
if $(n_1 == 0)$
        return
else
        $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
        $q_2 = BinarySearch(T[q_1], T, p_2, r_2)$
        $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
        $A[q_3] = T[q_1]$
        spawn $Par-Merge(T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3)$
        $Par-Merge(T, q_1 + 1, r_1, q_2 + 1, r_2, A, q_3 + 1)$
        sync
```

The Final Code for Parallel Merge

```
$Par - Merge(T, p_1, r_1, p_2, r_2, A, p_3)$
$n_1 = r_1 - p_1 + 1,\ n_2 = r_2 - p_2 + 1$
if $n_1 < n_2$
        Exchange $p_1 \leftrightarrow p_2, r_1 \leftrightarrow r_2, n_1 \leftrightarrow n_2$
if $(n_1 == 0)$
        return
else
        $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
        $q_2 = BinarySearch(T[q_1], T, p_2, r_2)$
        $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
        $A[q_3] = T[q_1]$
        spawn $Par - Merge(T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3)$
        $Par - Merge(T, q_1 + 1, r_1, q_2 + 1, r_2, A, q_3 + 1)$
        sync
```

The Final Code for Parallel Merge

```
$Par-Merge\left(T, p_1, r_1, p_2, r_2, A, p_3\right)$
$n_1 = r_1 - p_1 + 1$, $n_2 = r_2 - p_2 + 1$
if $n_1 < n_2$
        Exchange $p_1 \leftrightarrow p_2, r_1 \leftrightarrow r_2, n_1 \leftrightarrow n_2$
if $(n_1 == 0)$
        return
else
        $q_1 = \lfloor (p_1+r_1)/2 \rfloor$
        $q_2 = BinarySearch\left(T\left[q_1\right], T, p_2, r_2\right)$
        $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
        $A\left[q_3\right] = T\left[q_1\right]$
        spawn $Par-Merge\left(T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3\right)$
        $Par-Merge\left(T, q_1 + 1, r_1, q_2 + 1, r_2, A, q_3 + 1\right)$
        sync
```

The Final Code for Parallel Merge

```
$Par - Merge\left(T, p_1, r_1, p_2, r_2, A, p_3\right)$
    $n_1 = r_1 - p_1 + 1$, $n_2 = r_2 - p_2 + 1$
    if $n_1 < n_2$
        Exchange $p_1 \leftrightarrow p_2, r_1 \leftrightarrow r_2, n_1 \leftrightarrow n_2$
    if $(n_1 == 0)$
        return
    else
        $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
        $q_2 = BinarySearch\left(T\left[q_1\right], T, p_2, r_2\right)$
        $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
        $A\left[q_3\right] = T\left[q_1\right]$
        spawn $Par - Merge\left(T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3\right)$
        $Par - Merge\left(T, q_1 + 1, r_1, q_2 + 1, r_2, A, q_3 + 1\right)$
        sync
```

Explanation

Line 1

Obtain the length of the two arrays to be merged

Line 2: If one is larger than the other

We exchange the variables to work the largest element!!! In this case we make $n_1 \geq n_2$

Line 4

if $n_1 == 0$ return nothing to merge!!!

Explanation

Line 1

Obtain the length of the two arrays to be merged

Line 2: If one is larger than the other

We exchange the variables to work the largest element!!! In this case we make $n_1 \geq n_2$

Line 4

if $n_1 == 0$ return nothing to merge!!!

Explanation

Line 1

Obtain the length of the two arrays to be merged

Line 2: If one is larger than the other

We exchange the variables to work the largest element!!! In this case we make $n_1 \geq n_2$

Line 4

if $n_1 == 0$ return nothing to merge!!!

Explanation

Line 10

It copies $T[q_1]$ directly into $A[q_3]$

Line 11 and 12

They are used to recurse using nested parallelism to merge the sub-arrays less and greater than x.

Line 13

The sync is used to ensure that the subproblems have completed before the procedure returns.

Explanation

Line 10

It copies $T[q_1]$ directly into $A[q_3]$

Line 11 and 12

They are used to recurse using nested parallelism to merge the sub-arrays less and greater than x.

Line 13

The sync is used to ensure that the subproblems have completed before the procedure returns.

Explanation

Line 10

It copies $T[q_1]$ directly into $A[q_3]$

Line 11 and 12

They are used to recurse using nested parallelism to merge the sub-arrays less and greater than x.

Line 13

The sync is used to ensure that the subproblems have completed before the procedure returns.

First the Span Complexity of **Parallel Merge**: $T_\infty(n)$

Suppositions

- $n = n_1 + n_2$

What case should we study?

Remember $T_\infty(n) = \max\{T_\infty(n_1) + T_\infty(n_2)\}$

We notice then that

Because lines 3-6 $n_2 \leq n_1$

First the Span Complexity of **Parallel Merge**: $T_\infty(n)$

Suppositions

- $n = n_1 + n_2$

What case should we study?

Remember $T_\infty(n) = \max\{T_\infty(n_1) + T_\infty(n_2)\}$

We notice then that

Because lines 3-6 $n_2 \leq n_1$

First the Span Complexity of **Parallel Merge**: $T_\infty(n)$

Suppositions

- $n = n_1 + n_2$

What case should we study?

Remember $T_\infty(n) = \max\{T_\infty(n_1) + T_\infty(n_2)\}$

We notice then that

Because lines 3-6 $n_2 \leq n_1$

Span Complexity of the **Parallel Merge** with One Processor: $T_1(n)$

Then

$$2n_2 \leq n_1 + n_2 = n \Longrightarrow n_2 \leq {}^{n}/_{2}$$

Span Complexity of the **Parallel Merge** with One Processor: $T_1(n)$

Then

$$2n_2 \leq n_1 + n_2 = n \Longrightarrow n_2 \leq {}^{n}/_{2}$$

Thus

In the worst case, a recursive call in lines 11 merges:

- $\left\lfloor \frac{n_1}{2} \right\rfloor$ elements of $T\left[p_1 \ldots r_1\right]$ (Remember we are halving the array by mid-point).
- With all n_2 elements of $T\left[p_2 \ldots r_2\right]$.

Span Complexity of the **Parallel Merge** with One Processor: $T_1(n)$

Then

$$2n_2 \leq n_1 + n_2 = n \Longrightarrow n_2 \leq {}^{n}/_{2}$$

Thus

In the worst case, a recursive call in lines 11 merges:

- $\left\lfloor \frac{n_1}{2} \right\rfloor$ elements of $T[p_1 ... r_1]$ (Remember we are halving the array by mid-point).
- With all n_2 elements of $T[p_2 ... r_2]$.

Span Complexity of the **Parallel Merge** with One Processor: $T_1(n)$

Then

$$2n_2 \leq n_1 + n_2 = n \Longrightarrow n_2 \leq {}^{n}/_{2}$$

Thus

In the worst case, a recursive call in lines 11 merges:

- $\left\lfloor \frac{n_1}{2} \right\rfloor$ elements of $T\left[p_1 \ldots r_1\right]$ (Remember we are halving the array by mid-point).
- With all n_2 elements of $T\left[p_2 \ldots r_2\right]$.

Span Complexity of the **Parallel Merge** with One Processor: $T_1(n)$

Thus, the number of elements involved in such a call is

$$
\begin{aligned}
\left\lfloor \frac{n_1}{2} \right\rfloor + n_2 &\leq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n_2}{2} \\
&\leq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n/2}{2} \\
&= \frac{n_1 + n_2}{2} + \frac{n}{4} \\
&\leq \frac{n}{2} + \frac{n}{4} = \frac{3n}{4}
\end{aligned}
$$

Span Complexity of the **Parallel Merge** with One Processor: $T_1(n)$

Thus, the number of elements involved in such a call is

$$
\begin{aligned}
\left\lfloor \frac{n_1}{2} \right\rfloor + n_2 &\leq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n_2}{2} \\
&\leq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n/2}{2} \\
&= \frac{n_1 + n_2}{2} + \frac{n}{4} \\
&\leq \frac{n}{2} + \frac{n}{4} = \frac{3n}{4}
\end{aligned}
$$

Cinvestav

Span Complexity of the **Parallel Merge** with One Processor: $T_1(n)$

Thus, the number of elements involved in such a call is

$$\begin{aligned}\left\lfloor \frac{n_1}{2} \right\rfloor + n_2 &\leq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n_2}{2} \\ &\leq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n/2}{2} \\ &= \frac{n_1 + n_2}{2} + \frac{n}{4} \\ &\leq \frac{n}{2} + \frac{n}{4} = \frac{3n}{4}\end{aligned}$$

Span Complexity of the **Parallel Merge** with One Processor: $T_1(n)$

Thus, the number of elements involved in such a call is

$$\begin{aligned}
\left\lfloor \frac{n_1}{2} \right\rfloor + n_2 &\leq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n_2}{2} \\
&\leq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n/2}{2} \\
&= \frac{n_1 + n_2}{2} + \frac{n}{4} \\
&\leq \frac{n}{2} + \frac{n}{4} = \frac{3n}{4}
\end{aligned}$$

Cinvestav

Span Complexity of the **Parallel Merge** with One Processor: $T_1(n)$

Knowing that the Binary Search takes

$$\Theta(\log n)$$

We get the span for parallel merge

$$T_\infty(n) = T_\infty\left(\frac{3n}{4}\right) + \Theta(\log n)$$

This can can be solved using the exercise 4.6-2 in the Cormen's Book

$$T_\infty(n) = \Theta\left(\log^2 n\right)$$

Span Complexity of the **Parallel Merge** with One Processor: $T_1(n)$

Knowing that the Binary Search takes

$$\Theta(\log n)$$

We get the span for parallel merge

$$T_\infty(n) = T_\infty\left(\frac{3n}{4}\right) + \Theta(\log n)$$

This can can be solved using the exercise 4.6-2 in the Cormen's Book

$$T_\infty(n) = \Theta\left(\log^2 n\right)$$

Cinvestav

Span Complexity of the **Parallel Merge** with One Processor: $T_1(n)$

Knowing that the Binary Search takes

$$\Theta(\log n)$$

We get the span for parallel merge

$$T_\infty(n) = T_\infty\left(\frac{3n}{4}\right) + \Theta(\log n)$$

This can can be solved using the exercise 4.6-2 in the Cormen's Book

$$T_\infty(n) = \Theta\left(\log^2 n\right)$$

Calculating Work Complexity of **Parallel Merge**

Ok!!! We need to calculate the WORK

$$T_1(n) = \Theta(Something)$$

Thus

We need to calculate the upper and lower bound.

Cinvestav

Calculating Work Complexity of **Parallel Merge**

Ok!!! We need to calculate the WORK

$$T_1(n) = \Theta(Something)$$

Thus

We need to calculate the upper and lower bound.

Calculating Work Complexity of **Parallel Merge**

Work of Parallel Merge

The work of $T_1(n)$ of this Parallel Merge satisfies:

$$T_1(n) = \Omega(n)$$

- Because each of the n elements must be copied from array T to array A.

Calculating Work Complexity of **Parallel Merge**

Work of Parallel Merge

The work of $T_1(n)$ of this Parallel Merge satisfies:

$$T_1(n) = \Omega(n)$$

Because each of the n elements must be copied from array T to array A.

What about the Upper Bound O?

First notice that we can have a merge with

- $\frac{n}{4}$ elements when we have we have the worst case of $\left\lfloor \frac{n_1}{2} \right\rfloor + n_2$ in the other merge.
- And $\frac{3n}{4}$ for the worst case.
- And the work of the Binary Search of $O(\log n)$

Calculating Work Complexity of **Parallel Merge**

Work of Parallel Merge

The work of $T_1(n)$ of this Parallel Merge satisfies:

$$T_1(n) = \Omega(n)$$

Because each of the n elements must be copied from array T to array A.

What about the Upper Bound O?

First notice that we can have a merge with

- $\frac{n}{4}$ elements when we have we have the worst case of $\left\lfloor \frac{n_1}{2} \right\rfloor + n_2$ in the other merge.
- And $\frac{3n}{4}$ for the worst case.
- And the work of the Binary Search of $O(\log n)$

Calculating Work Complexity of **Parallel Merge**

Work of Parallel Merge

The work of $T_1(n)$ of this Parallel Merge satisfies:

$$T_1(n) = \Omega(n)$$

Because each of the n elements must be copied from array T to array A.

What about the Upper Bound O?

First notice that we can have a merge with

- $\frac{n}{4}$ elements when we have we have the worst case of $\left\lfloor \frac{n_1}{2} \right\rfloor + n_2$ in the other merge.
- And $\frac{3n}{4}$ for the worst case.
- And the work of the Binary Search of $O(\log n)$

Calculating Work Complexity of **Parallel Merge**

Work of Parallel Merge

The work of $T_1(n)$ of this Parallel Merge satisfies:

$$T_1(n) = \Omega(n)$$

Because each of the n elements must be copied from array T to array A.

What about the Upper Bound O?

First notice that we can have a merge with

- $\frac{n}{4}$ elements when we have we have the worst case of $\left\lfloor \frac{n_1}{2} \right\rfloor + n_2$ in the other merge.
- And $\frac{3n}{4}$ for the worst case.
- And the work of the Binary Search of $O(\log n)$

Calculating Work Complexity of **Parallel Merge**

Work of Parallel Merge

The work of $T_1\left(n\right)$ of this Parallel Merge satisfies:

$$T_1\left(n\right) = \Omega\left(n\right)$$

Because each of the n elements must be copied from array T to array A.

What about the Upper Bound O?

First notice that we can have a merge with

- $\frac{n}{4}$ elements when we have we have the worst case of $\left\lfloor\frac{n_1}{2}\right\rfloor + n_2$ in the other merge.
- And $\frac{3n}{4}$ for the worst case.
- And the work of the Binary Search of $O\left(\log n\right)$

Calculating Work Complexity of **Parallel Merge**

Work of Parallel Merge

The work of $T_1(n)$ of this Parallel Merge satisfies:

$$T_1(n) = \Omega(n)$$

Because each of the n elements must be copied from array T to array A.

What about the Upper Bound O?

First notice that we can have a merge with

- $\frac{n}{4}$ elements when we have we have the worst case of $\left\lfloor \frac{n_1}{2} \right\rfloor + n_2$ in the other merge.
- And $\frac{3n}{4}$ for the worst case.
- And the work of the Binary Search of $O(\log n)$

Calculating Work Complexity of **Parallel Merge**

Then

Then, for some $\alpha \in \left[\frac{1}{4}, \frac{3}{4}\right]$, then we have the following recursion for the Parallel Merge when we have one processor:

$$T_1(n) = \underbrace{T_1(\alpha n) + T_1((1-\alpha)n)}_{\text{Merge Part}} + \underbrace{\Theta(\log n)}_{\text{Binary Search}}$$

Remark: α varies at each level of the recursion!!!

Calculating Work Complexity of **Parallel Merge**

Then

Then, for some $\alpha \in \left[\frac{1}{4}, \frac{3}{4}\right]$, then we have the following recursion for the Parallel Merge when we have one processor:

$$T_1(n) = \underbrace{T_1(\alpha n) + T_1((1-\alpha)n)}_{\text{Merge Part}} + \underbrace{\Theta(\log n)}_{\text{Binary Search}}$$

Remark: α varies at each level of the recursion!!!

Calculating Work Complexity of **Parallel Merge**

Then

Then, for some $\alpha \in \left[\frac{1}{4}, \frac{3}{4}\right]$, then we have the following recursion for the Parallel Merge when we have one processor:

$$T_1(n) = \underbrace{T_1(\alpha n) + T_1((1-\alpha)n)}_{\text{Merge Part}} + \underbrace{\Theta(\log n)}_{\text{Binary Search}}$$

Remark: α varies at each level of the recursion!!!

Calculating Work Complexity of **Parallel Merge**

Then

Assume that $T_1(n) \leq c_1 n - c_2 \log n$ for positive constants c_1 and c_2.

We have then using c_3 for $\Theta(\log n)$

$$T_1(n) \leq T_1(\alpha n) + T_1((1-\alpha)n) + c_3 \log n$$

Calculating Work Complexity of **Parallel Merge**

Then

Assume that $T_1(n) \leq c_1 n - c_2 \log n$ for positive constants c_1 and c_2.

We have then using c_3 for $\Theta(\log n)$

$$\begin{aligned} T_1(n) &\leq T_1(\alpha n) + T_1((1-\alpha) n) + c_3 \log n \\ &\leq c_1 \alpha n - c_2 \log(\alpha n) + c_1 (1-\alpha) n - c_2 \log((1-\alpha) n) + c_3 \log n \\ &= c_1 n - c_2 \log(\alpha(1-\alpha)) - 2 c_2 \log n + c_3 \log n \text{ (splitting elements)} \\ &= c_1 n - c_2 (\log n + \log(\alpha(1-\alpha))) - (c_2 - c_3) \log n \\ &\leq c_1 n - (c_2 - c_3) \log n \text{ because } \log n + \log(\alpha(1-\alpha)) > 0 \end{aligned}$$

Cinvestav

Calculating Work Complexity of **Parallel Merge**

Then

Assume that $T_1(n) \leq c_1 n - c_2 \log n$ for positive constants c_1 and c_2.

We have then using c_3 for $\Theta(\log n)$

$$
\begin{aligned}
T_1(n) &\leq T_1(\alpha n) + T_1((1-\alpha)n) + c_3 \log n \\
&\leq c_1 \alpha n - c_2 \log(\alpha n) + c_1 (1-\alpha) n - c_2 \log((1-\alpha)n) + c_3 \log n \\
&= c_1 n - c_2 \log(\alpha(1-\alpha)) - 2 c_2 \log n + c_3 \log n \text{ (splitting elements)} \\
&= c_1 n - c_2 (\log n + \log(\alpha(1-\alpha))) - (c_2 - c_3) \log n \\
&\leq c_1 n - (c_2 - c_3) \log n \text{ because } \log n + \log(\alpha(1-\alpha)) > 0
\end{aligned}
$$

Calculating Work Complexity of **Parallel Merge**

Then

Assume that $T_1(n) \leq c_1 n - c_2 \log n$ for positive constants c_1 and c_2.

We have then using c_3 for $\Theta(\log n)$

$$
\begin{aligned}
T_1(n) &\leq T_1(\alpha n) + T_1((1-\alpha)n) + c_3 \log n \\
&\leq c_1 \alpha n - c_2 \log(\alpha n) + c_1 (1-\alpha) n - c_2 \log((1-\alpha)n) + c_3 \log n \\
&= c_1 n - c_2 \log(\alpha(1-\alpha)) - 2c_2 \log n + c_3 \log n \text{ (splitting elements)} \\
&= c_1 n - c_2 (\log n + \log(\alpha(1-\alpha))) - (c_2 - c_3) \log n \\
&\leq c_1 n - (c_2 - c_3) \log n \text{ because } \log n + \log(\alpha(1-\alpha)) > 0
\end{aligned}
$$

Calculating Work Complexity of **Parallel Merge**

Then

Assume that $T_1(n) \leq c_1 n - c_2 \log n$ for positive constants c_1 and c_2.

We have then using c_3 for $\Theta(\log n)$

$$\begin{aligned}
T_1(n) &\leq T_1(\alpha n) + T_1((1-\alpha)n) + c_3 \log n \\
&\leq c_1 \alpha n - c_2 \log(\alpha n) + c_1 (1-\alpha) n - c_2 \log((1-\alpha)n) + c_3 \log n \\
&= c_1 n - c_2 \log(\alpha(1-\alpha)) - 2c_2 \log n + c_3 \log n \text{ (splitting elements)} \\
&= c_1 n - c_2 (\log n + \log(\alpha(1-\alpha))) - (c_2 - c_3) \log n \\
&\leq c_1 n - (c_2 - c_3) \log n \text{ because } \log n + \log(\alpha(1-\alpha)) > 0
\end{aligned}$$

Calculating Work Complexity of **Parallel Merge**

Now, we have that given $0 < \alpha(1-\alpha) < 1$

We have $\log\left(\alpha(1-\alpha)\right) < 0$

Thus, making n large enough

$$\log n + \log\left(\alpha(1-\alpha)\right) > 0 \tag{1}$$

Then

$$T_1\left(n\right) \leq c_1 n - \left(c_2 - c_3\right) \log n$$

Calculating Work Complexity of **Parallel Merge**

Now, we have that given $0 < \alpha(1-\alpha) < 1$

We have $\log(\alpha(1-\alpha)) < 0$

Thus, making n large enough

$$\log n + \log(\alpha(1-\alpha)) > 0 \tag{1}$$

Then

$$T_1(n) \le c_1 n - (c_2 - c_3)\log n$$

Cinvestav

Calculating Work Complexity of **Parallel Merge**

Now, we have that given $0 < \alpha(1-\alpha) < 1$

We have $\log\left(\alpha(1-\alpha)\right) < 0$

Thus, making n large enough

$$\log n + \log\left(\alpha(1-\alpha)\right) > 0 \qquad (1)$$

Then

$$T_1(n) \leq c_1 n - (c_2 - c_3) \log n$$

Calculating Work Complexity of **Parallel Merge**

Now, we choose c_2 and c_3 such that

$$c_2 - c_3 \geq 0$$

We have that

$$T_1(n) \leq c_1 n = O(n)$$

Calculating Work Complexity of **Parallel Merge**

Now, we choose c_2 and c_3 such that

$$c_2 - c_3 \geq 0$$

We have that

$$T_1(n) \leq c_1 n = O(n)$$

Finally

Then

$$T_1(n) = \Theta(n)$$

The parallelism of Parallel Merge

$$\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{n}{\log^2 n}\right)$$

Finally

Then

$$T_1(n) = \Theta(n)$$

The parallelism of Parallel Merge

$$\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{n}{\log^2 n}\right)$$

Then What is the Complexity of Parallel Merge-sort with Parallel Merge?

First, the new code - Input $A[p..r]$ - Output $B[s..s+r-p]$

```
$Par-Merge-Sort(A,p,r,B,s)$
    $n = r - p + 1$
    if $(n == 1)$
            $B[s] = A[p]$
    else let $T[1..n]$ be a new array
            $q = \lfloor (p+r)/2 \rfloor$
            $q = q - p + 1$
            spawn $Par-Merge-Sort(A,p,q,T,1)$
            $Par-Merge-Sort(A,q+1,r,T,q'+1)$
            sync
            $Par-Merge(T,1,q',q'+1,n,B,s)$
```

Then What is the Complexity of Parallel Merge-sort with Parallel Merge?

First, the new code - Input $A\left[p..r\right]$ - Output $B\left[s..s+r-p\right]$

$Par-Merge-Sort\left(A,p,r,B,s\right)$

```
$n = r - p + 1$
if $(n == 1)$
        $B[s] = A[p]$
else let $T[1..n]$ be a new array
        $q = \lfloor (p+r)/2 \rfloor$
        $q = q - p + 1$
        spawn $Par-Merge-Sort(A, p, q, T, 1)$
        $Par-Merge-Sort(A, q+1, r, T, q'+1)$
        sync
        $Par-Merge(T, 1, q', q'+1, n, B, s)$
```

Then What is the Complexity of Parallel Merge-sort with Parallel Merge?

First, the new code - Input $A[p..r]$ - Output $B[s..s+r-p]$

$Par-Merge-Sort(A,p,r,B,s)$

```
$n = r - p + 1$
if $(n == 1)$
        $B[s] = A[p]$
else let $T[1..n]$ be a new array
        $q = \lfloor (p+r)/2 \rfloor$
        $q = q - p + 1$
        spawn $Par-Merge-Sort(A,p,q,T,1)$
        $Par-Merge-Sort(A,q+1,r,T,q'+1)$
        sync
        $Par-Merge(T,1,q',q'+1,n,B,s)$
```

Then What is the Complexity of Parallel Merge-sort with Parallel Merge?

First, the new code - Input $A\left[p..r\right]$ - Output $B\left[s..s+r-p\right]$

```
$Par-Merge-Sort\left(A,p,r,B,s\right)$
$n=r-p+1$
if $\left(n==1\right)$
        $B\left[s\right]=A\left[p\right]$
else let $T\left[1..n\right]$ be a new array
        $q=\left\lfloor (p+r)/2 \right\rfloor$
        $q=q-p+1$
        spawn $Par-Merge-Sort\left(A,p,q,T,1\right)$
        $Par-Merge-Sort\left(A,q+1,r,T,q'+1\right)$
        sync
        $Par-Merge\left(T,1,q',q'+1,n,B,s\right)$
```

Then What is the Complexity of Parallel Merge-sort with Parallel Merge?

First, the new code - Input $A\left[p..r\right]$ - Output $B\left[s..s+r-p\right]$

```
$Par-Merge-Sort\left(A,p,r,B,s\right)$
    $n=r-p+1$
    if $\left(n==1\right)$
        $B\left[s\right]=A\left[p\right]$
    else let $T\left[1..n\right]$ be a new array
        $q=\left\lfloor {}^{(p+r)}/_{2}\right\rfloor$
        $q=q-p+1$
        spawn $Par-Merge-Sort\left(A,p,q,T,1\right)$
        $Par-Merge-Sort\left(A,q+1,r,T,q'+1\right)$
        sync
        $Par-Merge\left(T,1,q',q'+1,n,B,s\right)$
```

Then What is the Complexity of Parallel Merge-sort with Parallel Merge?

First, the new code - Input $A\left[p..r\right]$ - Output $B\left[s..s+r-p\right]$

$Par-Merge-Sort\left(A,p,r,B,s\right)$

```
$n = r - p + 1$
if $(n == 1)$
        $B[s] = A[p]$
else let $T[1..n]$ be a new array
        $q = \lfloor (p+r)/2 \rfloor$
        $q = q - p + 1$
        spawn $Par-Merge-Sort(A, p, q, T, 1)$
        $Par-Merge-Sort(A, q+1, r, T, q'+1)$
        sync
        $Par-Merge(T, 1, q', q'+1, n, B, s)$
```

Then, What is the amount of Parallelism of Parallel Merge-sort with Parallel Merge?

Work

We can use the worst work in the parallel to generate the recursion:

$$
\begin{aligned}
T_1^{PMS}(n) &= 2T_1^{PMS}\left(\frac{n}{2}\right) + T_1^{PM}(n) \\
&= 2T_1^{PMS}\left(\frac{n}{2}\right) + \Theta(n) \\
&= \Theta(n \log n) \text{ Case 2 of the MT}
\end{aligned}
$$

Then, What is the amount of Parallelism of Parallel Merge-sort with Parallel Merge?

Work

We can use the worst work in the parallel to generate the recursion:

$$T_1^{PMS}(n) = 2\,T_1^{PMS}\left(\frac{n}{2}\right) + T_1^{PM}(n)$$

$$= 2\,T_1^{PMS}\left(\frac{n}{2}\right) + \Theta(n)$$

$$= \Theta(n \log n) \text{ Case 2 of the MT}$$

Then, What is the amount of Parallelism of Parallel Merge-sort with Parallel Merge?

Work

We can use the worst work in the parallel to generate the recursion:

$$\begin{aligned} T_1^{PMS}(n) &= 2T_1^{PMS}\left(\frac{n}{2}\right) + T_1^{PM}(n) \\ &= 2T_1^{PMS}\left(\frac{n}{2}\right) + \Theta(n) \\ &= \Theta(n \log n) \text{ Case 2 of the MT} \end{aligned}$$

Cinvestav

Then, What is the amount of Parallelism of Parallel Merge-sort with Parallel Merge?

Work

We can use the worst work in the parallel to generate the recursion:

$$\begin{aligned} T_1^{PMS}(n) &= 2T_1^{PMS}\left(\frac{n}{2}\right) + T_1^{PM}(n) \\ &= 2T_1^{PMS}\left(\frac{n}{2}\right) + \Theta(n) \\ &= \Theta(n\log n) \text{ Case 2 of the MT} \end{aligned}$$

Then, What is the amount of Parallelism of Parallel Merge-sort with Parallel Merge?

Span

We get the following recursion for the span by taking in account that lines 7 and 8 of parallel merge sort run in parallel:

$$
\begin{aligned}
T_{\infty}^{PMS}(n) &= T_{\infty}^{PMS}\left(\frac{n}{2}\right) + T_{\infty}^{PM}(n) \\
&= T_{\infty}^{PMS}\left(\frac{n}{2}\right) + \Theta\left(\log^2 n\right) \\
&= \Theta\left(\log^3 n\right) \text{ Exercise 4.6-2 in the Cormen's Book}
\end{aligned}
$$

Then, What is the amount of Parallelism of Parallel Merge-sort with Parallel Merge?

Span

We get the following recursion for the span by taking in account that lines 7 and 8 of parallel merge sort run in parallel:

$$\begin{aligned} T_{\infty}^{PMS}(n) &= T_{\infty}^{PMS}\left(\frac{n}{2}\right) + T_{\infty}^{PM}(n) \\ &= T_{\infty}^{PMS}\left(\frac{n}{2}\right) + \Theta\left(\log^2 n\right) \\ &= \Theta\left(\log^3 n\right) \text{ Exercise 4.6-2 in the Cormen's Book} \end{aligned}$$

Then, What is the amount of Parallelism of Parallel Merge-sort with Parallel Merge?

Span

We get the following recursion for the span by taking in account that lines 7 and 8 of parallel merge sort run in parallel:

$$\begin{aligned} T_{\infty}^{PMS}(n) &= T_{\infty}^{PMS}\left(\frac{n}{2}\right) + T_{\infty}^{PM}(n) \\ &= T_{\infty}^{PMS}\left(\frac{n}{2}\right) + \Theta\left(\log^2 n\right) \\ &= \Theta\left(\log^3 n\right) \text{ Exercise 4.6-2 in the Cormen's Book} \end{aligned}$$

Cinvestav

Then, What is the amount of Parallelism of Parallel Merge-sort with Parallel Merge?

Span

We get the following recursion for the span by taking in account that lines 7 and 8 of parallel merge sort run in parallel:

$$\begin{aligned} T_{\infty}^{PMS}(n) &= T_{\infty}^{PMS}\left(\frac{n}{2}\right) + T_{\infty}^{PM}(n) \\ &= T_{\infty}^{PMS}\left(\frac{n}{2}\right) + \Theta\left(\log^2 n\right) \\ &= \Theta\left(\log^3 n\right) \text{ Exercise 4.6-2 in the Cormen's Book} \end{aligned}$$

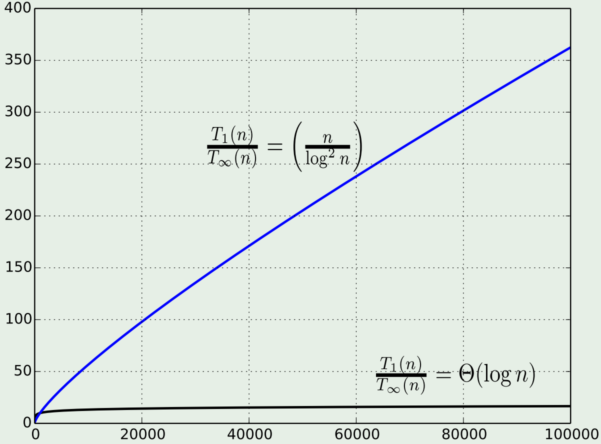
Then, What is the amount of Parallelism of Parallel Merge-sort with Parallel Merge?

Parallelism

$$\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{n}{\log^2 n}\right)$$

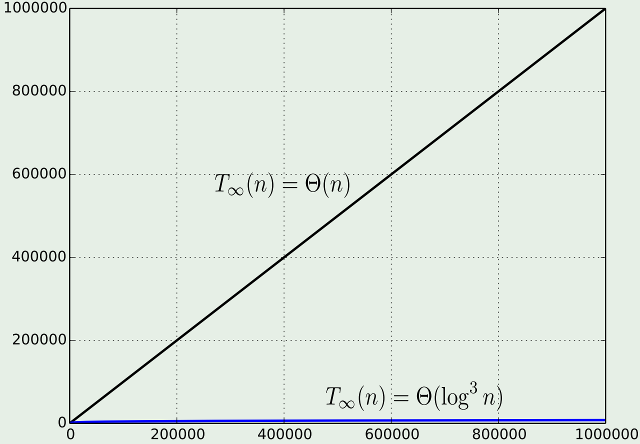
Plotting both Parallelisms

We get the incredible difference between both algorithm



Plotting the T_∞

We get the incredible difference when running both algorithms with an infinite number of processors!!!



Outline

Exercises

- 27.1-1
- 27.1-2
- 27.1-4
- 27.1-6
- 27.1-7
- 27.2-1
- 27.2-3
- 27.2-4
- 27.2-5
- 27.3-1
- 27.3-2
- 27.3-3
- 27.3-4