Parallelism

Petr Kurapov

Fall 2024

Agenda

- Recursion tree and master theorem
- Quantifying parallelism
- For-loop analysis
- Parallel merge sort analysis

Recursion tree and master theorem

- Asymptotic algorithm behavior
- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$$f(n)$$

$$T\left(\frac{n}{b}\right) \quad T\left(\frac{n}{b}\right) \quad T\left(\frac{n}{b}\right)$$

Recursion tree and master theorem

- Asymptotic algorithm behavior
- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$$f(n)$$

$$f\left(\frac{n}{b}\right) \quad f\left(\frac{n}{b}\right) \quad f\left(\frac{n}{b}\right)$$

$$T\left(\frac{n}{b^2}\right) \quad T\left(\frac{n}{b^2}\right)$$

Recursion tree and master theorem

- Asymptotic algorithm behavior
- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$f(n)$

$f\left(\frac{n}{b}\right)$ $f\left(\frac{n}{b}\right)$ $f\left(\frac{n}{b}\right)$

$T\left(\frac{n}{b^2}\right)$ $T\left(\frac{n}{b^2}\right)$

$T(1)$

Recursion tree and master theorem

- Asymptotic algorithm behavior
- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$f(n)$	$f(n)$
$f\left(\frac{n}{b}\right) \quad f\left(\frac{n}{b}\right) \quad f\left(\frac{n}{b}\right)$	$af\left(\frac{n}{b}\right)$
$T\left(\frac{n}{b^2}\right) \quad T\left(\frac{n}{b^2}\right)$	$a^2T\left(\frac{n}{b^2}\right)$
$T(1)$	

Recursion tree and master theorem

- Asymptotic algorithm behavior
- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$f(n)$	$f(n)$
$f\left(\frac{n}{b}\right)$ $f\left(\frac{n}{b}\right)$ $f\left(\frac{n}{b}\right)$	$af\left(\frac{n}{b}\right)$
$T\left(\frac{n}{b^2}\right)$ $T\left(\frac{n}{b^2}\right)$	$a^2T\left(\frac{n}{b^2}\right)$
$T(1)$	$a^{\log_b n}T(1)$

Recursion tree and master theorem

- Asymptotic algorithm behavior
- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$$f(n)$$

$$f\left(\frac{n}{b}\right) \quad f\left(\frac{n}{b}\right) \quad f\left(\frac{n}{b}\right)$$

$$T\left(\frac{n}{b^2}\right) \quad T\left(\frac{n}{b^2}\right)$$

$$T(1)$$

$$f(n)$$

$$af\left(\frac{n}{b}\right)$$

$$a^2T\left(\frac{n}{b^2}\right)$$

$$a^{\log_b n}T(1)$$

$$\Theta\left(n^{\log_b a}\right)$$

Recursion tree and master theorem

- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
- Compare $f(n)$ to $n^{\log_b a}$
  - Leaf-heavy: $f(n) = O(n^{\log_b a - \varepsilon})$
    - $T(n) = \Theta(n^{\log_b a})$
  - Comparable: $f(n) = \Theta\left(n^{\log_b a} (\log n)^k\right)$
    - $T(n) = \Theta(n^{\log_b a} (\log n)^{k+1})$
  - Root-heavy: $f(n) = \Omega(n^{\log_b a + \varepsilon})$
    - $T(n) = \Theta(f(n))$
    - Regularity: a$f\left(\frac{n}{b}\right) \leq cf(n), c < 1$

$f(n)$

$f\left(\frac{n}{b}\right)$ $f\left(\frac{n}{b}\right)$ $f\left(\frac{n}{b}\right)$

$T\left(\frac{n}{b^2}\right)$ $T\left(\frac{n}{b^2}\right)$

$T(1)$

$f(n)$

$af\left(\frac{n}{b}\right)$

$a^2T\left(\frac{n}{b^2}\right)$

$a^{\log_b n}T(1)$

$\|$

$\Theta\left(n^{\log_b a}\right)$

QUIZ TIME

- $T(n) = 4T\left(\frac{n}{2}\right) + n$

Leaf-heavy: $f(n) = \ O(n^{\log_b a - \varepsilon})$

- $T(n) = n^{\log_b a}$

Comparable: $f(n) =$
$\Theta\left(n^{\log_b a} (\log n)^k\right)$

- $T(n) = n^{\log_b a} (\log n)^{k+1}$

Root-heavy: $f(n) = \Omega\ (n^{\log_b a + \varepsilon})$

- $T(n) = \Theta(f(n))$
- Regularity: a$f\left(\frac{n}{b}\right) \leq cf(n), c < 1$

QUIZ TIME

- $T(n) = 4T\left(\frac{n}{2}\right) + n$
    - $T(n) = \Theta(n^2)$

Leaf-heavy: $f(n) = \ O(n^{\log_b a - \varepsilon})$

- $T(n) = n^{\log_b a}$

Comparable: $f(n) = \Theta\left(n^{\log_b a} (\log n)^k\right)$

- $T(n) = n^{\log_b a} (\log n)^{k+1}$

Root-heavy: $f(n) = \Omega\ (n^{\log_b a + \varepsilon})$

- $T(n) = \Theta(f(n))$
- Regularity: a$f\left(\frac{n}{b}\right) \leq cf(n), c < 1$

QUIZ TIME

- $T(n) = 4T\left(\frac{n}{2}\right) + n$
  - $T(n) = \Theta(n^2)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

Leaf-heavy: $f(n) = \ O(n^{\log_b a - \varepsilon})$

- $T(n) = n^{\log_b a}$

Comparable: $f(n) =$
$\Theta\left(n^{\log_b a} (\log n)^k\right)$

- $T(n) = n^{\log_b a} (\log n)^{k+1}$

Root-heavy: $f(n) = \Omega\ (n^{\log_b a + \varepsilon})$

- $T(n) = \Theta(f(n))$
- Regularity: $af\left(\frac{n}{b}\right) \leq cf(n), c < 1$

QUIZ TIME

- $T(n) = 4T\left(\frac{n}{2}\right) + n$
  - $T(n) = \Theta(n^2)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2$
  - $T(n) = \Theta(n^2 \log n)$

Leaf-heavy: $f(n) = \; O(n^{\log_b a - \varepsilon})$

- $T(n) = n^{\log_b a}$

Comparable: $f(n) = \Theta\left(n^{\log_b a} (\log n)^k\right)$

- $T(n) = n^{\log_b a} (\log n)^{k+1}$

Root-heavy: $f(n) = \Omega\,(n^{\log_b a + \varepsilon})$

- $T(n) = \Theta(f(n))$
- Regularity: a$f\left(\frac{n}{b}\right) \leq cf(n), c < 1$

QUIZ TIME

- $T(n) = 4T\left(\frac{n}{2}\right) + n$
  - $T(n) = \Theta(n^2)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2$
  - $T(n) = \Theta(n^2 \log n)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

Leaf-heavy: $f(n) = \ O(n^{\log_b a - \varepsilon})$
- $T(n) = n^{\log_b a}$

Comparable: $f(n) = \Theta\left(n^{\log_b a} (\log n)^k\right)$
- $T(n) = n^{\log_b a} (\log n)^{k+1}$

Root-heavy: $f(n) = \Omega\ (n^{\log_b a + \varepsilon})$
- $T(n) = \Theta(f(n))$
- Regularity: a$f\left(\frac{n}{b}\right) \leq cf(n), c < 1$

QUIZ TIME

- $T(n) = 4T\left(\frac{n}{2}\right) + n$
  - $T(n) = \Theta(n^2)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2$
  - $T(n) = \Theta(n^2 \log n)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^3$
  - $T(n) = \Theta(n^3)$

Leaf-heavy: $f(n) = O(n^{\log_b a - \varepsilon})$

- $T(n) = n^{\log_b a}$

Comparable: $f(n) = \Theta\left(n^{\log_b a} (\log n)^k\right)$

- $T(n) = n^{\log_b a} (\log n)^{k+1}$

Root-heavy: $f(n) = \Omega\,(n^{\log_b a + \varepsilon})$

- $T(n) = \Theta(f(n))$
- Regularity: a$f\left(\frac{n}{b}\right) \leq cf(n), c < 1$

QUIZ TIME

- $T(n) = 4T\left(\frac{n}{2}\right) + n$
  - $T(n) = \Theta(n^2)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2$
  - $T(n) = \Theta(n^2 \log n)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^3$
  - $T(n) = \Theta(n^3)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2/\log n$

Leaf-heavy: $f(n) = O(n^{\log_b a - \varepsilon})$

- $T(n) = n^{\log_b a}$

Comparable: $f(n) = \Theta\left(n^{\log_b a} (\log n)^k\right)$

- $T(n) = n^{\log_b a} (\log n)^{k+1}$

Root-heavy: $f(n) = \Omega\, (n^{\log_b a + \varepsilon})$

- $T(n) = \Theta(f(n))$
- Regularity: $af\left(\frac{n}{b}\right) \leq cf(n), c < 1$

QUIZ TIME

- $T(n) = 4T\left(\frac{n}{2}\right) + n$
  - $T(n) = \Theta(n^2)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2$
  - $T(n) = \Theta(n^2 \log n)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^3$
  - $T(n) = \Theta(n^3)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2 / \log n$
  - $T(n) = \Theta(n^2 \log \log n)$

Leaf-heavy: $f(n) = O(n^{\log_b a - \varepsilon})$
- $T(n) = n^{\log_b a}$

Comparable: $f(n) = \Theta\left(n^{\log_b a} (\log n)^k\right)$
- $T(n) = n^{\log_b a} (\log n)^{k+1}$

Root-heavy: $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$
- $T(n) = \Theta(f(n))$
- Regularity: $af\left(\frac{n}{b}\right) \le cf(n), c < 1$

QUIZ TIME

- $T(n) = 4T\left(\frac{n}{2}\right) + n$
  - $T(n) = \Theta(n^2)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2$
  - $T(n) = \Theta(n^2 \log n)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^3$
  - $T(n) = \Theta(n^3)$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2/\log n$
  - $T(n) = \Theta(n^2 \log\log n)$

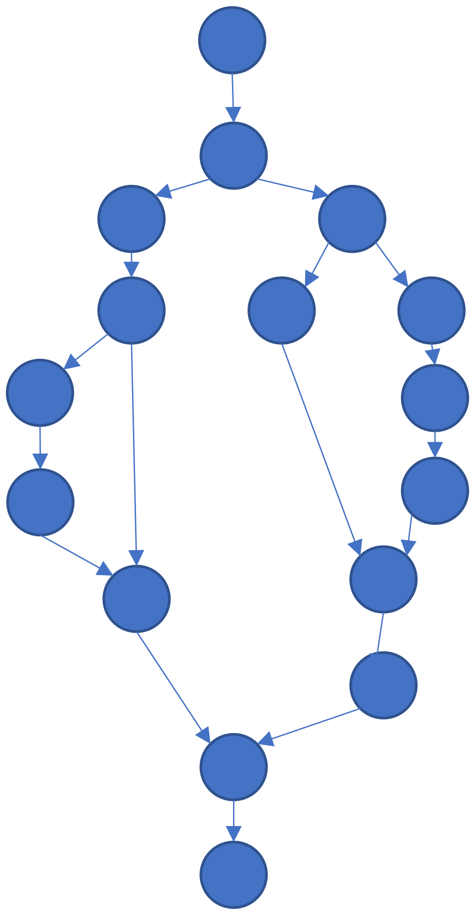
Akra-Bazzi method – for wider classes

Parallelism

How do you measure parallelism?

Parallelism

- Amdahl's law
  - $L_s = \frac{1}{(1-p)+p/s}$, p – parallel portion, s – speedup
  - $L \leq 1/\alpha$
  - Is this practical?



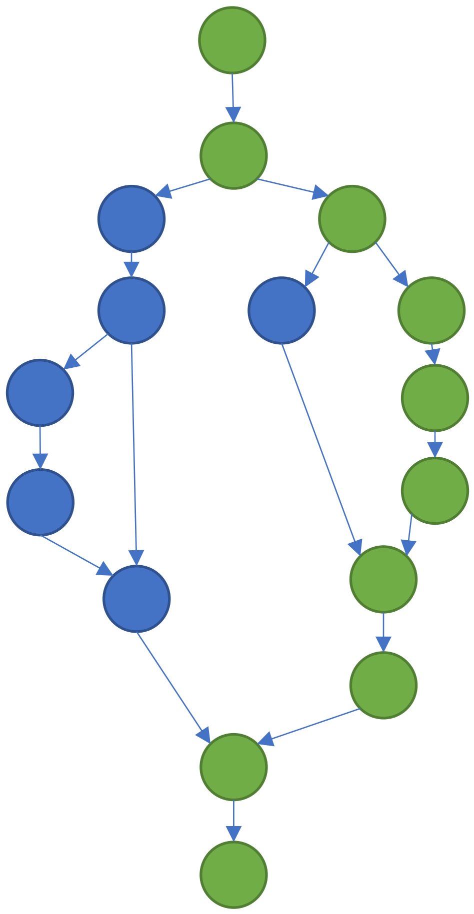
Parallelism

- Amdahl's law
  - $L_s = \frac{1}{(1-p)+p/s}$, p – parallel portion, s – speedup
  - $L \leq 1/\alpha$

T – total amount of work

T_p - execution time for P abstract exe units ($\geq T/T_p$)

T_{span} - critical path ($\leq T_p$)



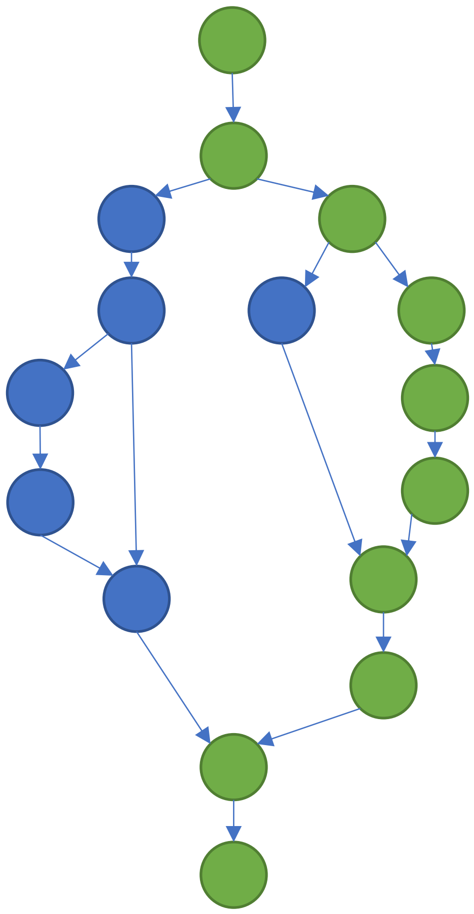
Parallelism

T – total amount of work

T_p - execution time for P abstract exe units ($\geq T/T_p$)

T_{span} - critical path ($\leq T_p$)

$Speedup = T/p$ (superlinear?)
$Parallelism = T/T_{span}$



Parallelism

Example:

```
int fib(int n) {
    return (n < 2)? n :
        fib(n-1) + fib(n-2);
}
```

How much parallelism does `fib(4)` have?

Parallelism

Example:

```
int fib(int n) {
    return (n < 2)? n :
      fib(n-1) + fib(n-2);
}
```

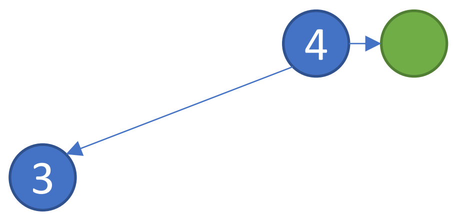
How much parallelism does `fib(4)` have?

Parallelism

Example:

```
int fib(int n) {
    return (n < 2)? n :
      fib(n-1) + fib(n-2);
}
```

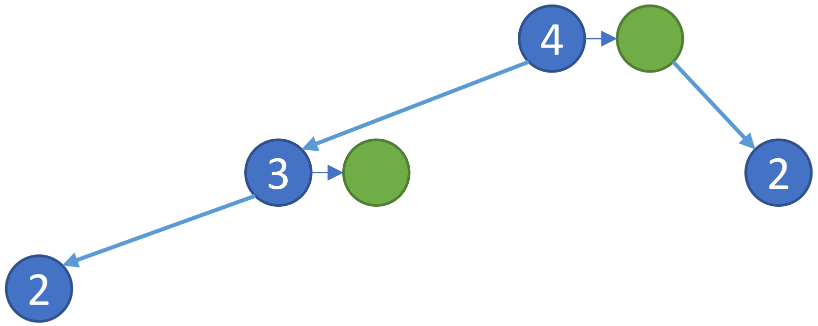
How much parallelism does `fib(4)` have?



Parallelism

Example:

```
int fib(int n) {
    return (n < 2)? n :
      fib(n-1) + fib(n-2);
}
```

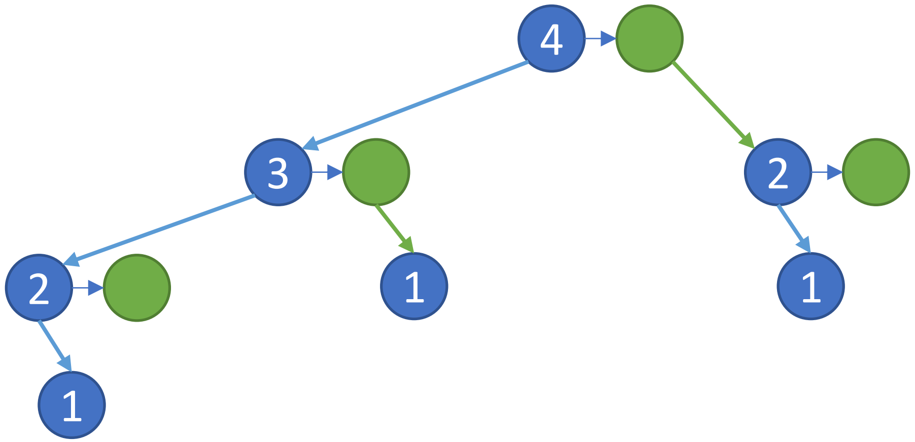


How much parallelism does `fib(4)` have?

Parallelism

Example:

```
int fib(int n) {
    return (n < 2)? n :
      fib(n-1) + fib(n-2);
}
```



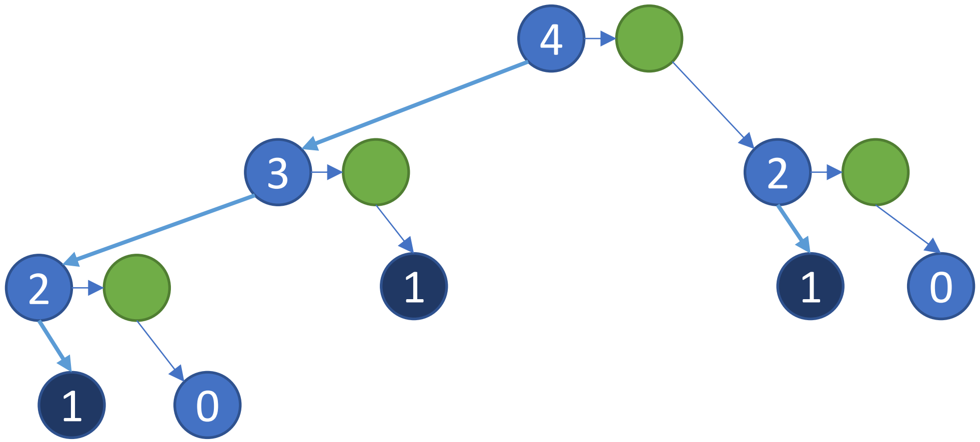
How much parallelism does `fib(4)` have?

Parallelism

Example:

```
int fib(int n) {
    return (n < 2)? n :
      fib(n-1) + fib(n-2);
}
```

How much parallelism does `fib(4)` have?

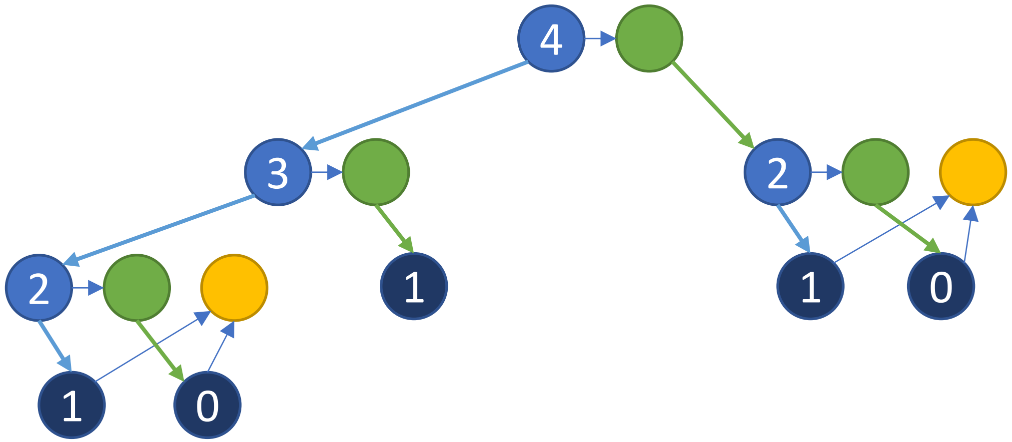


Parallelism

Example:

```
int fib(int n) {
    return (n < 2)? n :
      fib(n-1) + fib(n-2);
}
```

How much parallelism does `fib(4)` have?

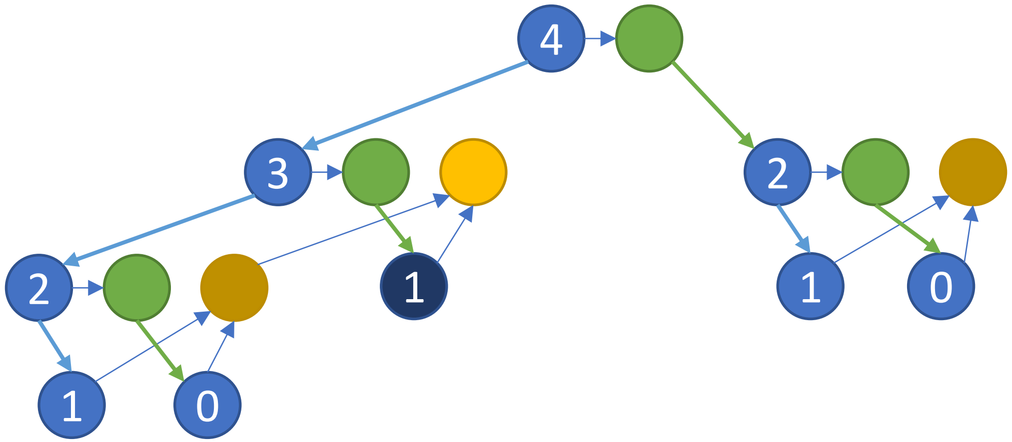


Parallelism

Example:

```
int fib(int n) {
    return (n < 2)? n :
      fib(n-1) + fib(n-2);
}
```

How much parallelism does `fib(4)` have?

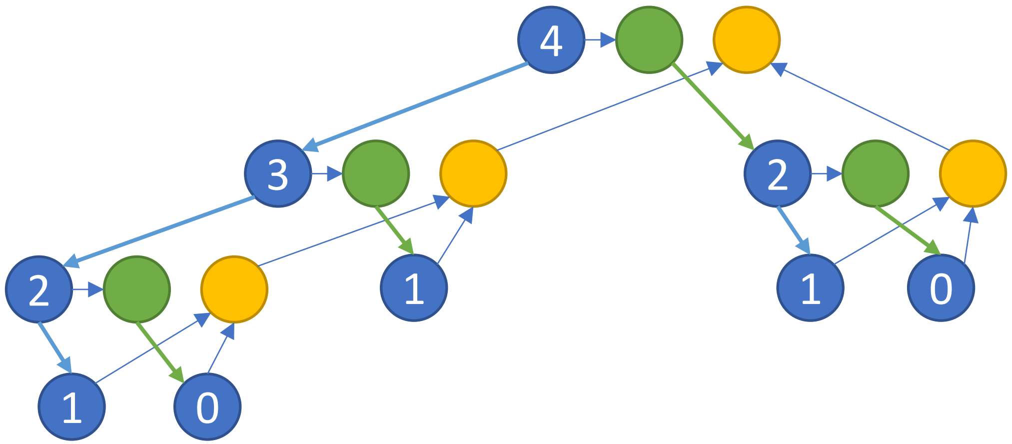


Parallelism

Example:

```
int fib(int n) {
    return (n < 2)? n :
      fib(n-1) + fib(n-2);
}
```

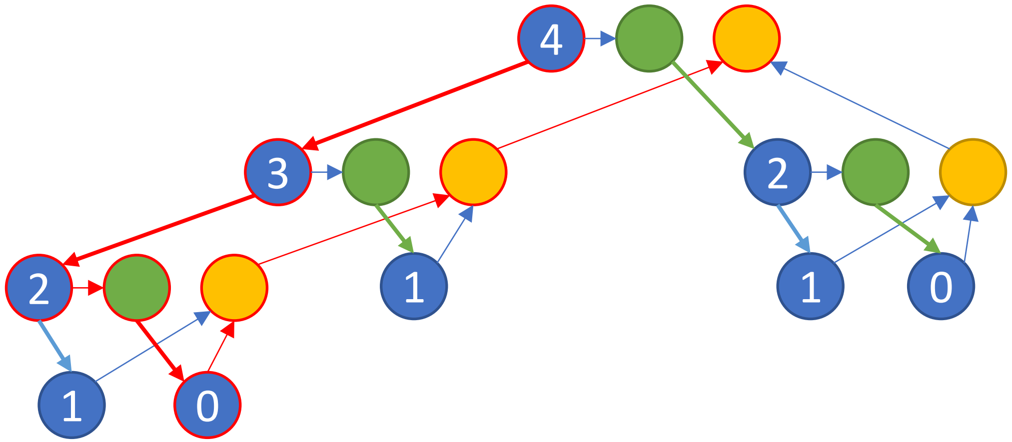
How much parallelism does `fib(4)` have?



Parallelism

Example:

```
int fib(int n) {
    return (n < 2)? n :
      fib(n-1) + fib(n-2);
}
```



How much parallelism does `fib(4)` have?

- 17/8 = 2.125

Parallel execution mode

- Dynamic multithreading vs static threading
  - Shared (distributed) memory
  - Concurrency platforms – resource planning and handling
  - `parallel_for, spawn, sync`
  - Cilk/Cilk++, OpenMP, Task Parallel Library, Threading Building Blocks
- Theoretical basis for parallel algorithm analysis
  - Span & work
- Recurrences and divide & conquer

For loops analysis

```
parallel_for(int i=1; i<sz; i++) {
    for (int j=0; j<i; j++) {
        swap(a[i][j], a[j][i]);
    }
}
```

- Most of work parallelization happens in loops
- Divide and conquer implementation

For loops analysis

```
parallel_for(int i=1; i<sz; i++) {
    for (int j=0; j<i; j++) {
        swap(a[i][j], a[j][i]);
    }
}
```

```
void control(int low, int high) {
    if (high > low + 1) {
        control(low, (high+low)/2);
        control((high+low)/2, high);
    }
    int i = low;
    for (int j = 0; j < i; j++)
        swap(a[i][j], a[j][i]);
}
```

For loops analysis

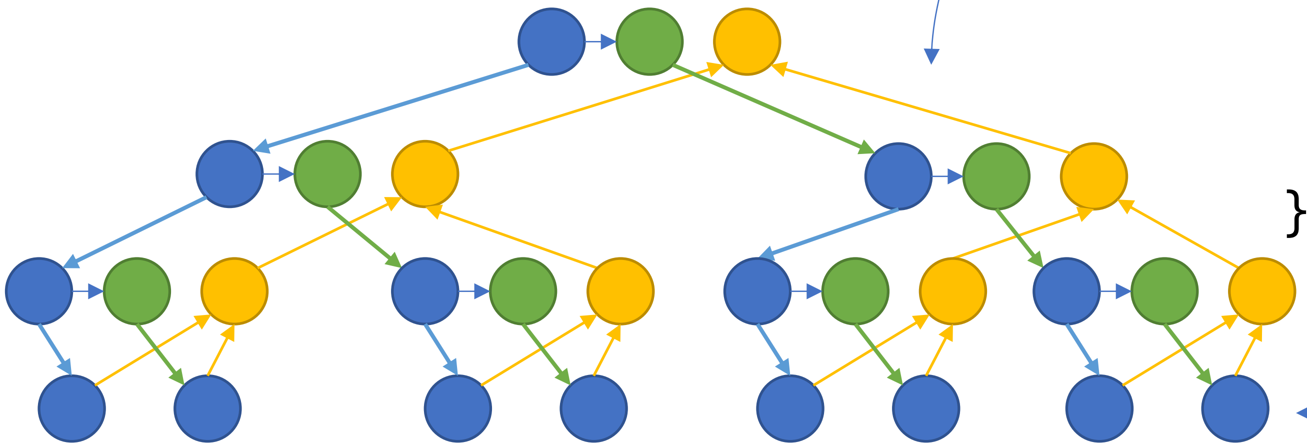
```
parallel_for(int i=1; i<sz; i++) {
    for (int j=0; j<i; j++) {
        swap(a[i][j], a[j][i]);
    }
}
```

```
void control(int low, int high) {
    if (high > low + 1) {
spawn   control(low, (high+low)/2);
        control((high+low)/2, high);
sync;}
    int i = low;
    for (int j = 0; j < i; j++)
        swap(a[i][j], a[j][i]);
}
```

For loops analysis

```
parallel_for(int i=1; i<sz; i++) {
    for (int j=0; j<i; j++) {
        swap(a[i][j], a[j][i]);
    }
}
```

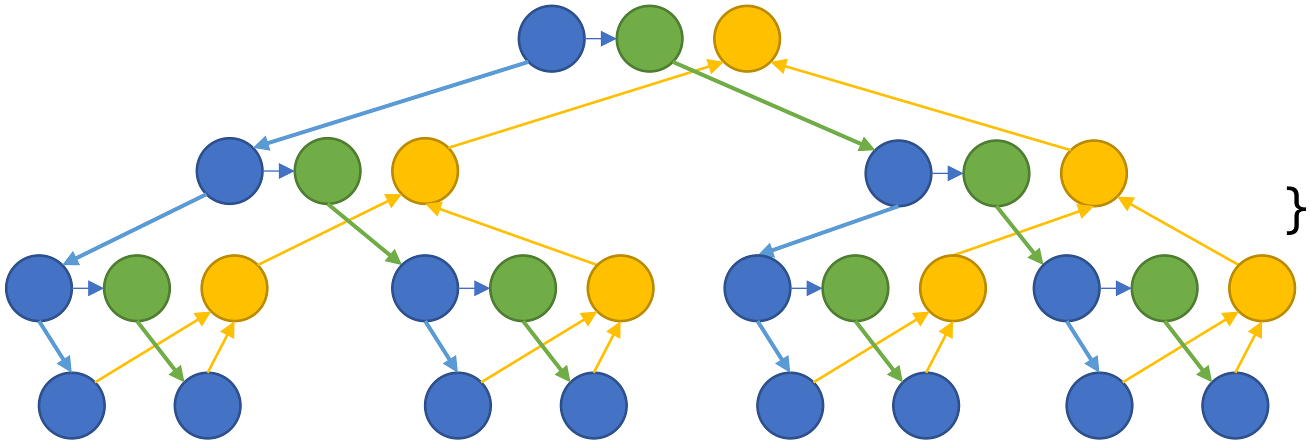
```
void control(int low, int high) {
    if (high > low + 1) {
spawn   control(low, (high+low)/2);
        control((high+low)/2, high);
sync;}
    int i = low;
    for (int j = 0; j < i; j++)
        swap(a[i][j], a[j][i]);
}
```



For loops analysis

```
parallel_for(int i=1; i<sz; i++) {
    for (int j=0; j<i; j++) {
        swap(a[i][j], a[j][i]);
    }
}
```

```
void control(int low, int high) {
    if (high > low + 1) {
spawn   control(low, (high+low)/2);
        control((high+low)/2, high);
sync;}
    int i = low;
    for (int j = 0; j < i; j++)
        swap(a[i][j], a[j][i]);
}
```

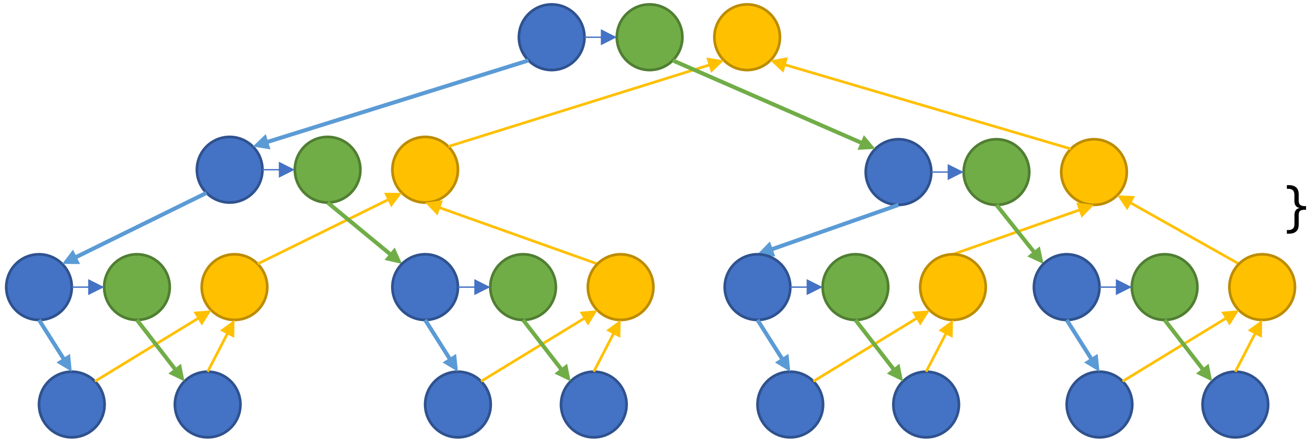


Work	
Span	
Parallelism	

For loops analysis

```
parallel_for(int i=1; i<sz; i++) {
    for (int j=0; j<i; j++) {
        swap(a[i][j], a[j][i]);
    }
}
```

```
void control(int low, int high) {
    if (high > low + 1) {
spawn   control(low, (high+low)/2);
        control((high+low)/2, high);
sync;}
    int i = low;
    for (int j = 0; j < i; j++)
        swap(a[i][j], a[j][i]);
}
```

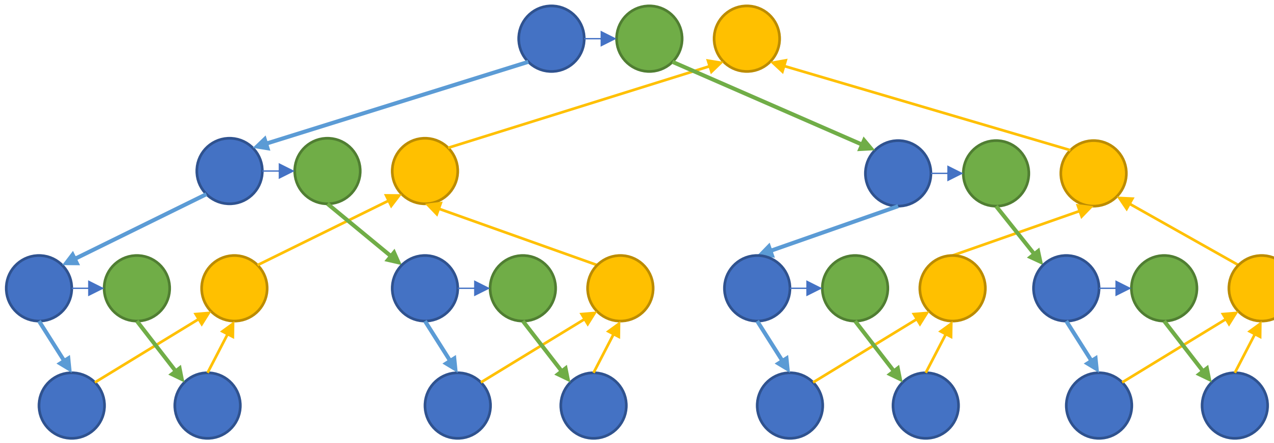


Work	$\Theta(n^2)$
Span	
Parallelism	

For loops analysis

```
parallel_for(int i=1; i<sz; i++) {
    for (int j=0; j<i; j++) {
        swap(a[i][j], a[j][i]);
    }
}
```

```
void control(int low, int high) {
    if (high > low + 1) {
spawn   control(low, (high+low)/2);
        control((high+low)/2, high);
sync;}
    int i = low;
    for (int j = 0; j < i; j++)
        swap(a[i][j], a[j][i]);
}
```

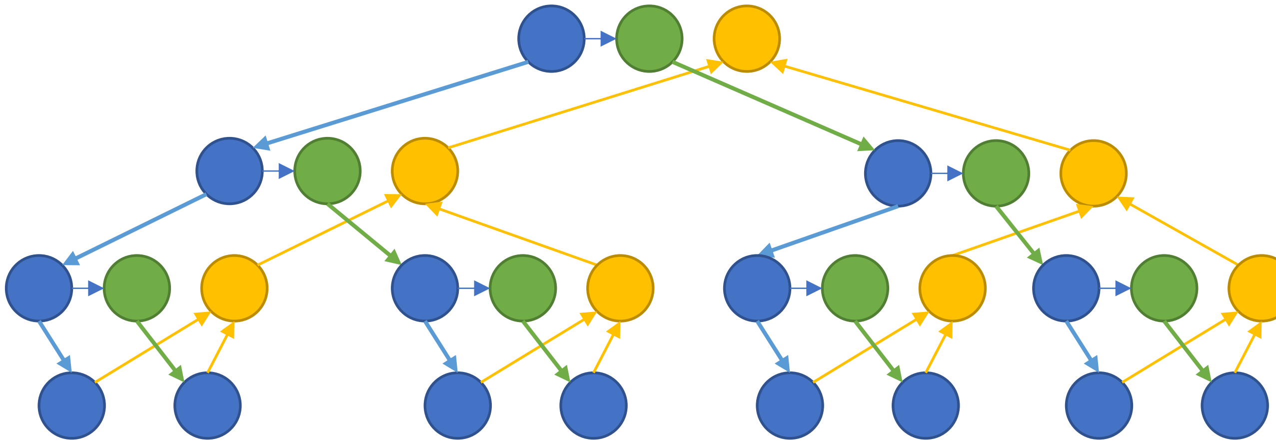


Work	$\Theta(n^2)$
Span	$\Theta(n) = \Theta(n + \log n)$
Parallelism	

For loops analysis

```
parallel_for(int i=1; i<sz; i++) {
    for (int j=0; j<i; j++) {
        swap(a[i][j], a[j][i]);
    }
}
```

```
void control(int low, int high) {
    if (high > low + 1) {
spawn   control(low, (high+low)/2);
        control((high+low)/2, high);
sync;}
    int i = low;
    for (int j = 0; j < i; j++)
        swap(a[i][j], a[j][i]);
}
```

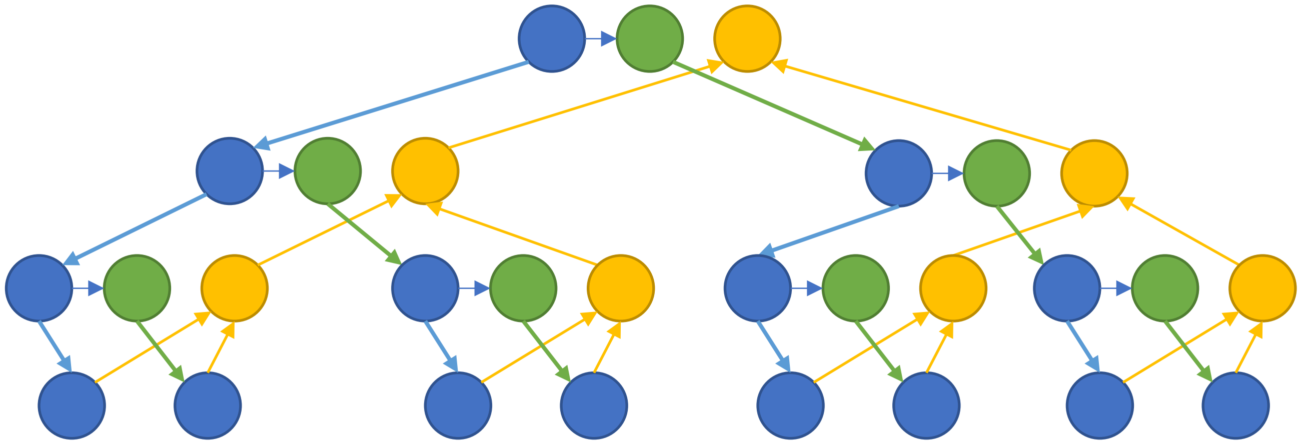


Work	$\Theta(n^2)$
Span	$\Theta(n) = \Theta(n + \log n)$
Parallelism	$\Theta(n)$

For loops analysis

```
parallel_for(int i=1; i<sz; i++) {
  parallel_for(int j=0; j<i; j++) {
        swap(a[i][j], a[j][i]);
    }
}
```

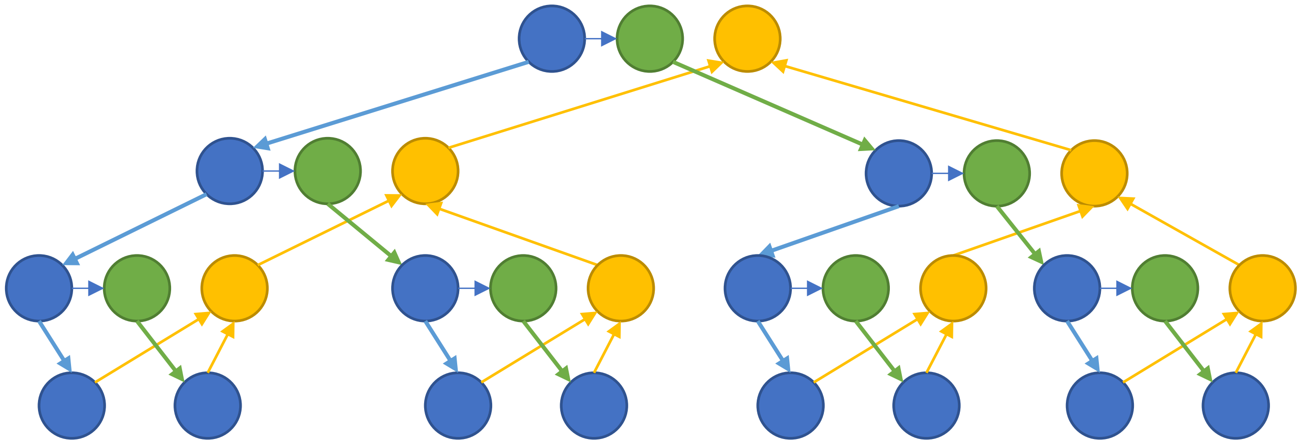
Work	$\Theta(n^2)$
Span	
Parallelism	



For loops analysis

```
parallel_for(int i=1; i<sz; i++) {
  parallel_for(int j=0; j<i; j++) {
        swap(a[i][j], a[j][i]);
    }
}
```

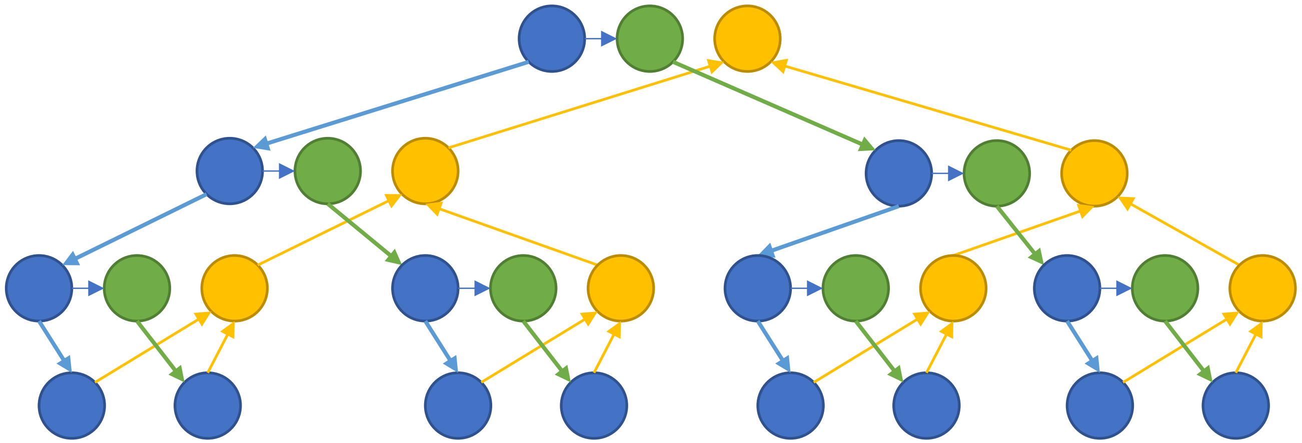
Work	$\Theta(n^2)$
Span	$\Theta(\log n)$
Parallelism	



For loops analysis

```
parallel_for(int i=1; i<sz; i++) {
  parallel_for(int j=0; j<i; j++) {
        swap(a[i][j], a[j][i]);
    }
}
```

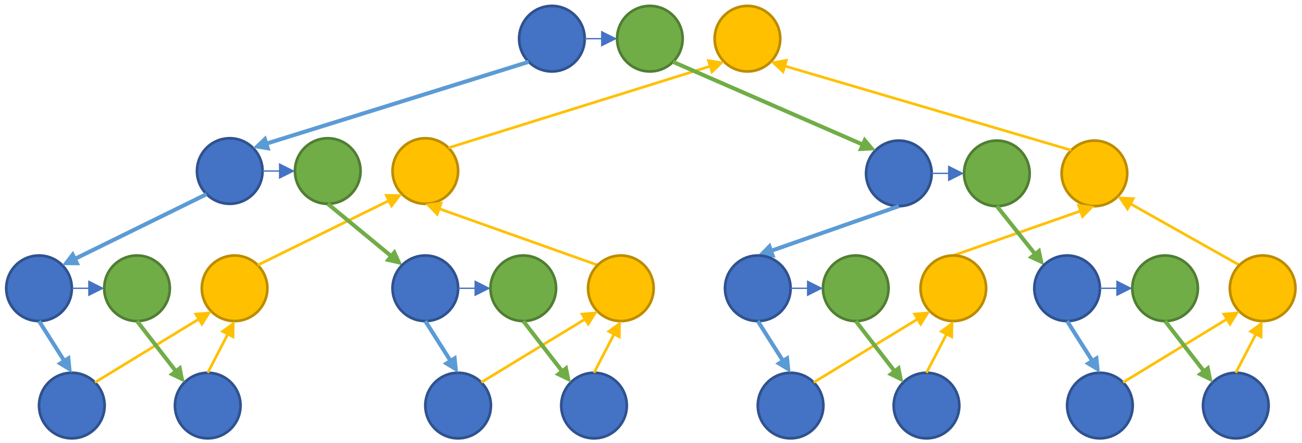
Work	$\Theta(n^2)$
Span	$\Theta(\log n)$
Parallelism	$\Theta(n^2/\log n)$



For loops analysis

- Coarsening
  - Motivation: control and scheduling overhead >> problem size

```
void control(int low, int high) {
    if (high > low + 1) {
spawn  control(low, (high+low)/2);
       control((high+low)/2, high);
sync;}
…
```

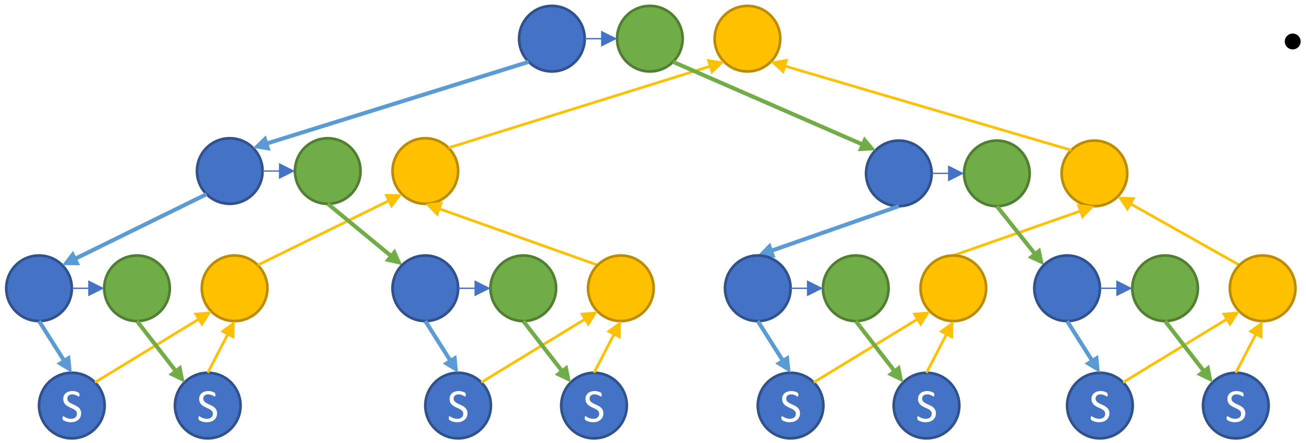


For loops analysis

- Coarsening
  - Motivation: control and scheduling overhead >> problem size
  - Split n-size problem into chunks

```
void control(int low, int high) {
    if (high > low + CHUNK_SIZE) {
spawn  control(low, (high+low)/2);
       control((high+low)/2, high);
sync;}
…
```

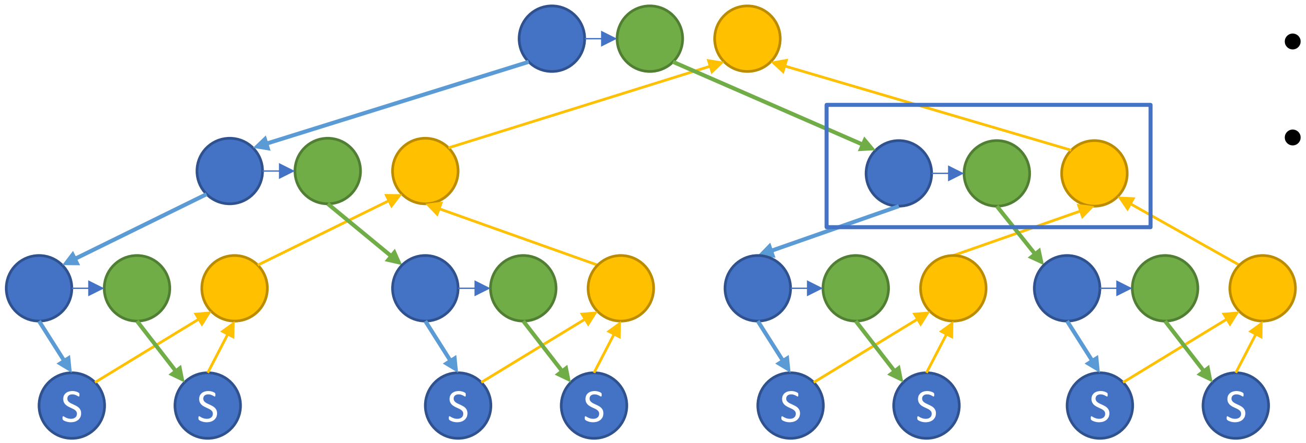
- Let `CHUNK_SIZE` = S, single item = I



For loops analysis

- Coarsening
  - Motivation: control and scheduling overhead >> problem size
  - Split n-size problem into chunks

```
void control(int low, int high) {
    if (high > low + CHUNK_SIZE) {
spawn  control(low, (high+low)/2);
       control((high+low)/2, high);
sync;}
…
```



- Let CHUNK_SIZE = S, single item = I
- Let control logic work = C (spawn, ret)

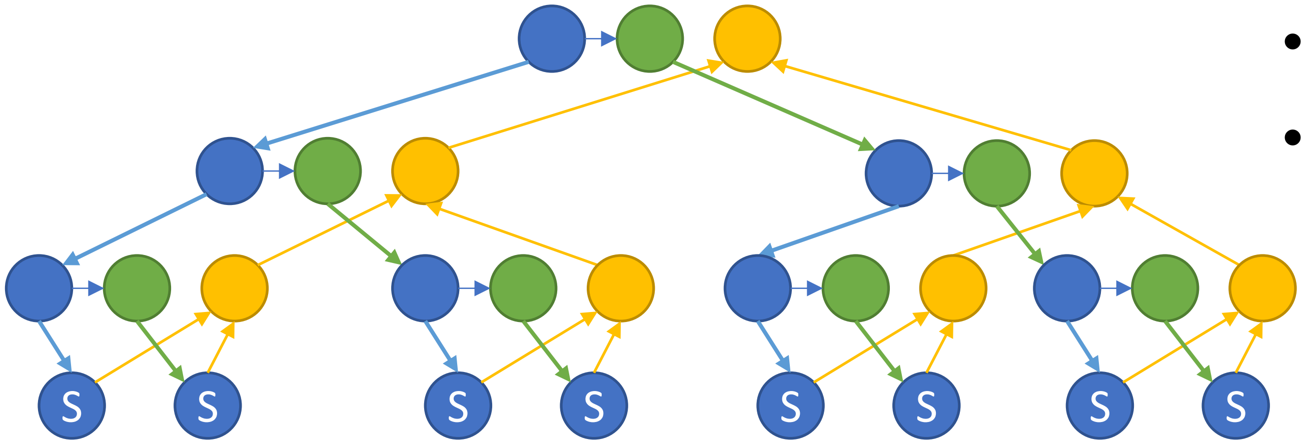
Work	
Span	

For loops analysis

- Coarsening
  - Motivation: control and scheduling overhead >> problem size
  - Split n-size problem into chunks

```
void control(int low, int high) {
    if (high > low + CHUNK_SIZE) {
spawn  control(low, (high+low)/2);
       control((high+low)/2, high);
sync;}
…
```

- Let CHUNK_SIZE = S, single item = I
- Let control logic work = C (spawn, ret)



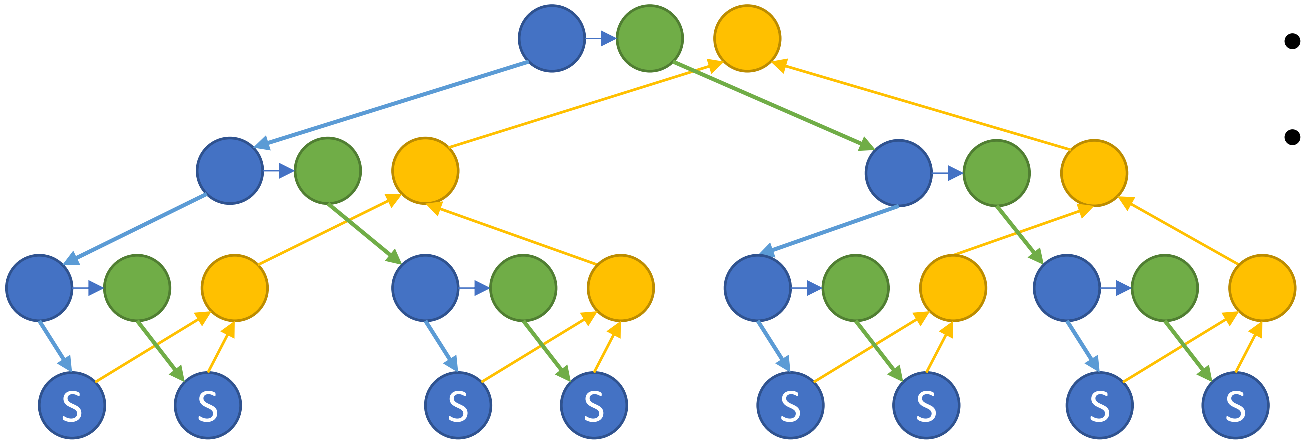
Work	$nI + (n/S - 1)C$
Span	

For loops analysis

- Coarsening
  - Motivation: control and scheduling overhead >> problem size
  - Split n-size problem into chunks

```
void control(int low, int high) {
    if (high > low + CHUNK_SIZE) {
spawn  control(low, (high+low)/2);
       control((high+low)/2, high);
sync;}
…
```

- Let CHUNK_SIZE = S, single item = I
- Let control logic work = C (spawn, ret)

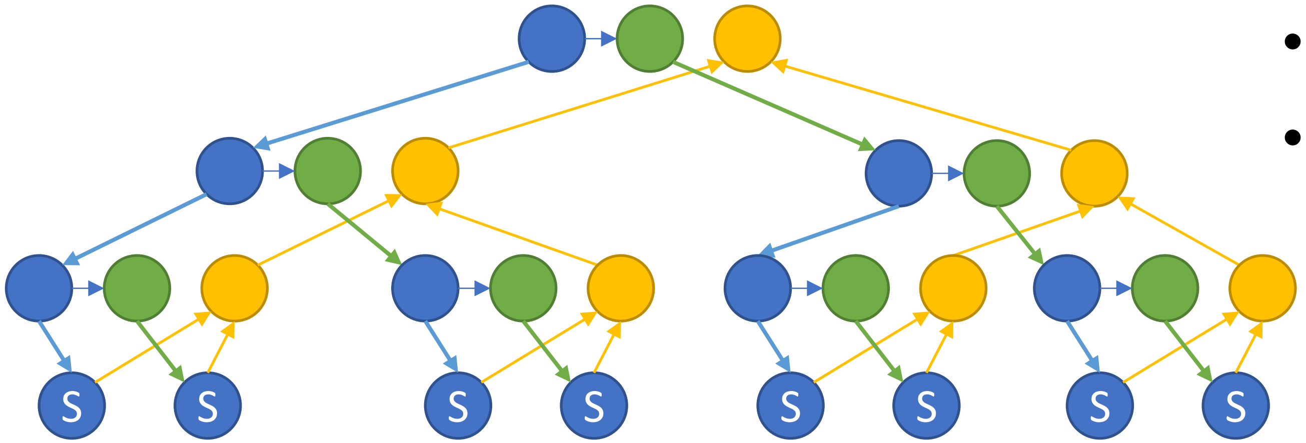


Work	$nI + (n/S - 1)C$
Span	$SI + C \log n/S$

For loops analysis

- Coarsening
  - Motivation: control and scheduling overhead >> problem size
  - Split n-size problem into chunks

```
void control(int low, int high) {
    if (high > low + CHUNK_SIZE) {
spawn  control(low, (high+low)/2);
       control((high+low)/2, high);
sync;}
…
```



- Let CHUNK_SIZE = S, single item = I
- Let control logic work = C (spawn, ret)

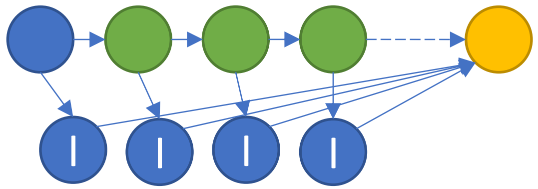
Work	$nI + (n/S - 1)C$
Span	$SI + C \log n/S$

S >> C/I

For loops analysis

- What happens to parallelism if workers are spawned in loop?

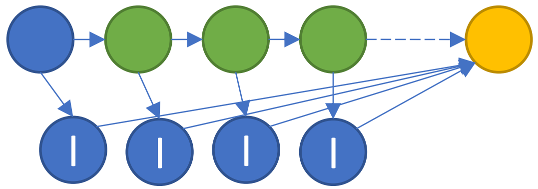
```
for (int i = 0; i < n; i++)
{
    spawn dosmth(i, i+1);
}
```



For loops analysis

- What happens to parallelism if workers are spawned in loop?
- $T(n) = \Theta(n)$
- $T_{span}(n) = \Theta(n)$
- Parallelism: $\Theta(1)$

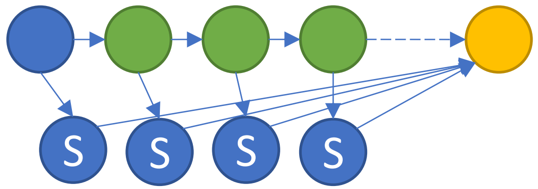
```
for (int i = 0; i < n; i++)
{
    spawn dosmth(i, i+1);
}
```



For loops analysis

- What happens to parallelism if workers are spawned in loop?
- Does coarsening help?
- $T(n) =$
- $T_{span}(n) =$

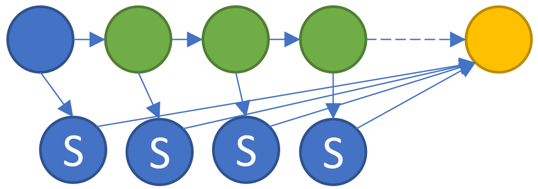
```
for (int i = 0; i < n; i+=S)
{
    spawn dosmth(i, i+S);
}
```



For loops analysis

- What happens to parallelism if workers are spawned in loop?
- Does coarsening help?
- $T(n) = \Theta(n)$
- $T_{span}(n) = \Theta(S + n/S)$

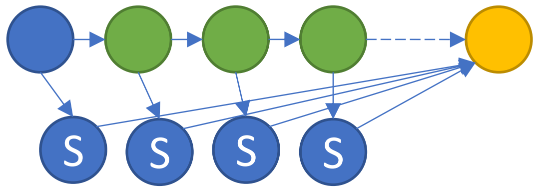
```
for (int i = 0; i < n; i+=S)
{
    spawn dosmth(i, i+S);
}
```



For loops analysis

- What happens to parallelism if workers are spawned in loop?
- Does coarsening help?
- $T(n) = \Theta(n)$
- $T_{span}(n) = \Theta(S + n/S)$
- Parallelism: $\Theta(\sqrt{n})$

```
for (int i = 0; i < n; i+=S)
{
    spawn dosmth(i, i+S);
}
```



Merge sort

```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        mergesort(a, p, q);
        mergesort(a, q+1, r);
        merge(a, p, q, r);
    }
}
```

```
void merge(int* a, int p, int q, int r){
    int n1 = q-p+1, n2 = r-q;
    int i = 1, j = 1;
    // lt[1..n1+1]=a[p+1..n1]
    // rt[1..n2+1]=a[n2..q+j]
    for (int k = p; k < r; ++k) {
        if (lt[i] <= rt[j])
            a[k] = lt[i]; i++;
        else
            a[k] = rt[j]; j++;
    }
}
```

Introduction to algorithms 3 ed. Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein

Merge sort

```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        spawn mergesort(a, p, q);
        mergesort(a, q+1, r);
        sync; merge(a, p, q, r);
    }
}
```

```
void merge(int* a, int p, int q, int r){
    int n1 = q-p+1, n2 = r-q;
    int i = 1, j = 1;
    // lt[1..n1+1]=a[p+1..n1]
    // rt[1..n2+1]=a[n2..q+j]
    for (int k = p; k < r; ++k) {
        if (lt[i] <= rt[j])
            a[k] = lt[i]; i++;
        else
            a[k] = rt[j]; j++;
    }
}
```

Merge sort

```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        spawn mergesort(a, p, q);
        mergesort(a, q+1, r);
        sync; merge(a, p, q, r);
    }
}
```

- Merge: $\Theta(n)$
- Sort: $T(n) = 2T(n/2) + \Theta(n)$

```
void merge(int* a, int p, int q, int r){
    int n1 = q-p+1, n2 = r-q;
    int i = 1, j = 1;
    // lt[1..n1+1]=a[p+1..n1]
    // rt[1..n2+1]=a[n2..q+j]
    for (int k = p; k < r; ++k) {
        if (lt[i] <= rt[j])
            a[k] = lt[i]; i++;
        else
            a[k] = rt[j]; j++;
    }
}
```

Merge sort

```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        spawn mergesort(a, p, q);
        mergesort(a, q+1, r);
        sync; merge(a, p, q, r);
    }
}
```

```
void merge(int* a, int p, int q, int r){
    int n1 = q-p+1, n2 = r-q;
    int i = 1, j = 1;
    // lt[1..n1+1]=a[p+1..n1]
    // rt[1..n2+1]=a[n2..q+j]
    for (int k = p; k < r; ++k) {
        if (lt[i] <= rt[j])
            a[k] = lt[i]; i++;
        else
            a[k] = rt[j]; j++;
    }
}
```

- Merge: $\Theta(n)$
- Sort: $T(n) = \Theta(n \log n)$

$$n^{\log_b a} = n, f(n) = n^{\log_b a} (\log n)^0$$

Merge sort

```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        spawn mergesort(a, p, q);
        mergesort(a, q+1, r);
        sync; merge(a, p, q, r);
    }
}
```

```
void merge(int* a, int p, int q, int r){
    int n1 = q-p+1, n2 = r-q;
    int i = 1, j = 1;
    // lt[1..n1+1]=a[p+1..n1]
    // rt[1..n2+1]=a[n2..q+j]
    for (int k = p; k < r; ++k) {
        if (lt[i] <= rt[j])
            a[k] = lt[i]; i++;
        else
            a[k] = rt[j]; j++;
    }
}
```

Work	$\Theta(n \log n)$
Span	
Parallelism	

Merge sort

```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        spawn mergesort(a, p, q);
        mergesort(a, q+1, r);
        sync; merge(a, p, q, r);
    }
}
```

Work	$\Theta(n \log n)$
Span	$T_\infty(n) = T(n/2) + \Theta(n)$
Parallelism	

```
void merge(int* a, int p, int q, int r){
    int n1 = q-p+1, n2 = r-q;
    int i = 1, j = 1;
    // lt[1..n1+1]=a[p+1..n1]
    // rt[1..n2+1]=a[n2..q+j]
    for (int k = p; k < r; ++k) {
        if (lt[i] <= rt[j])
            a[k] = lt[i]; i++;
        else
            a[k] = rt[j]; j++;
    }
}
```

Merge sort

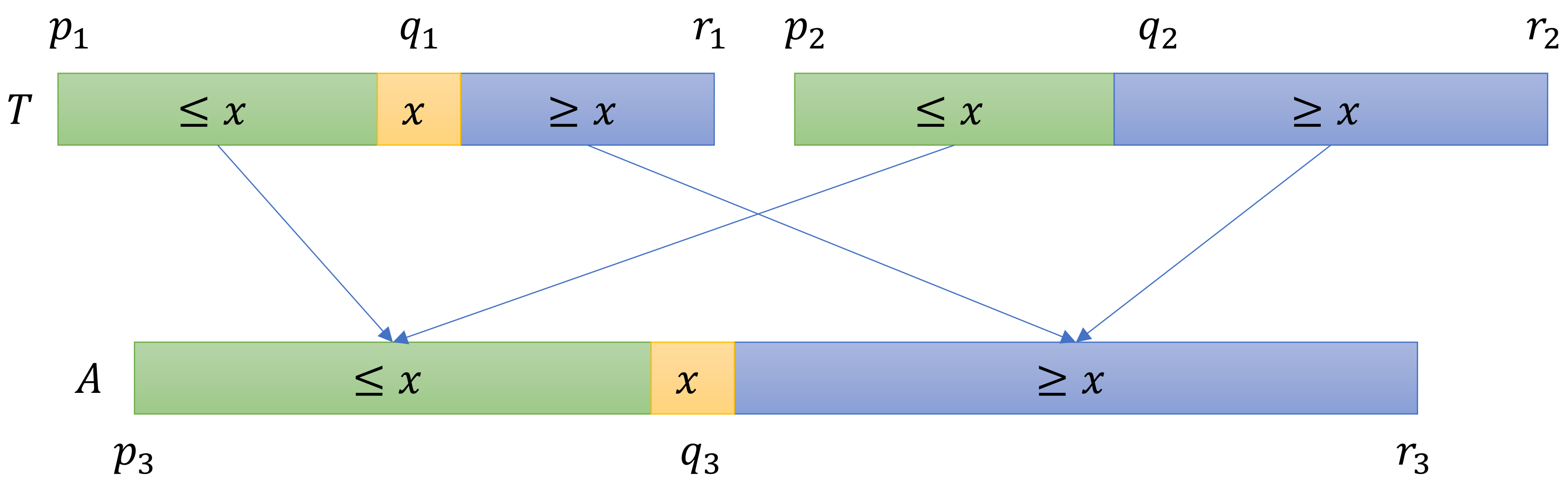
```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        spawn mergesort(a, p, q);
        mergesort(a, q+1, r);
        sync; merge(a, p, q, r);
    }
}
```

Work	$\Theta(n \log n)$
Span	$\Theta(n)$
Parallelism	$\Theta(\log n)$

```
void merge(int* a, int p, int q, int r){
    int n1 = q-p+1, n2 = r-q;
    int i = 1, j = 1;
    // lt[1..n1+1]=a[p+1..n1]
    // rt[1..n2+1]=a[n2..q+j]
    for (int k = p; k < r; ++k) {
        if (lt[i] <= rt[j])
            a[k] = lt[i]; i++;
        else
            a[k] = rt[j]; j++;
    }
}
```

Merge sort

- Parallel merge idea



$$x = T[q_1 = (p_1 + r_1)/2]$$

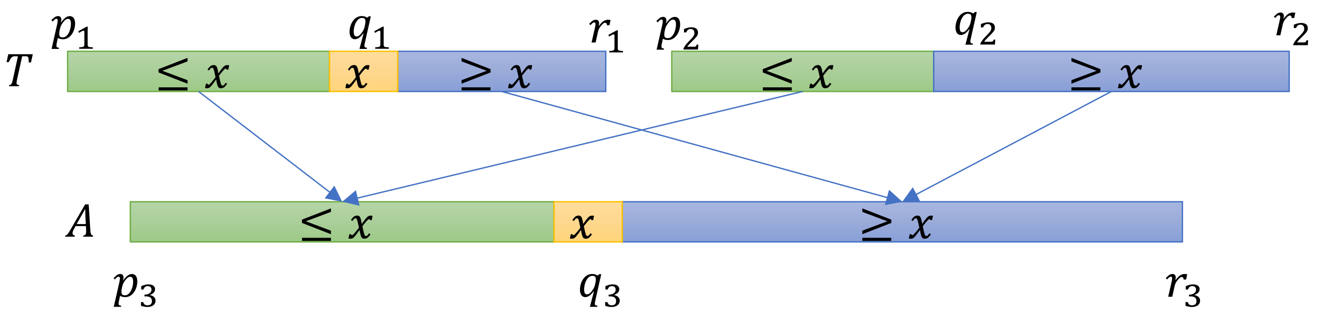
Merge sort

```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        spawn mergesort(a, p, q);
        mergesort(a, q+1, r);
        sync; pmerge(a, p, q, r);
    }
}
```

```
void pmerge(...)
{
```

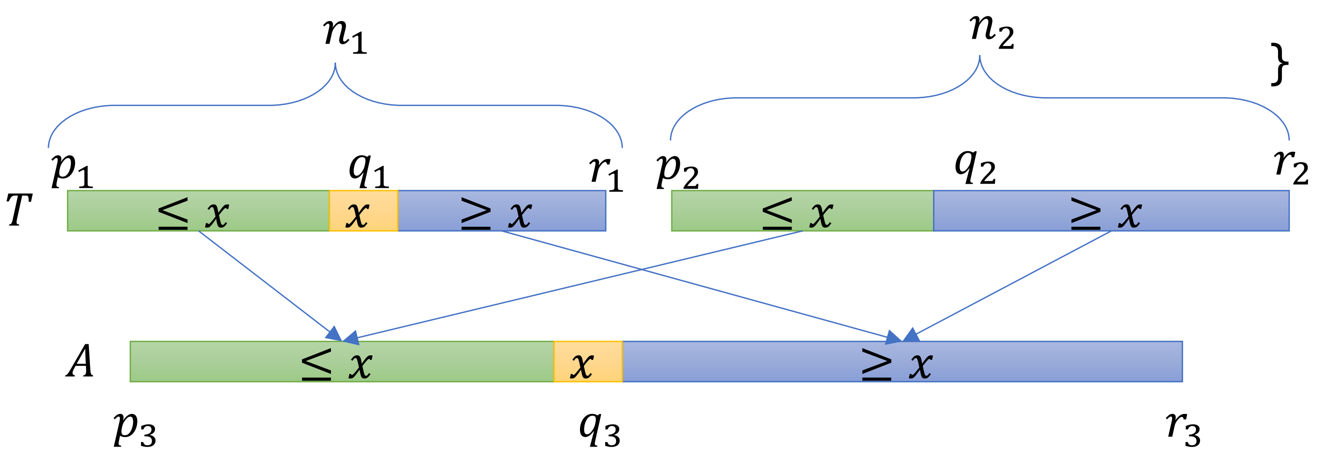
- Find mid point x in T_1, T_2 (binary search for 2) & copy to target
- Recursive merge T[p1..q1), T[p2..q2) & store to A[p3..q3)
- Recursive merge T(q1..r1], T(q2..r2] & store to A(q3..r3]

```
}
```



Merge sort

$$n = n_1 + n_2$$




```
void pmerge(...)
{
```

- Find mid point x in T_1, T_2 (binary search for 2) & copy to target
- Recursive merge T[p1..q1), T[p2..q2) & store to A[p3..q3)
- Recursive merge T(q1..r1], T(q2..r2] & store to A(q3..r3]

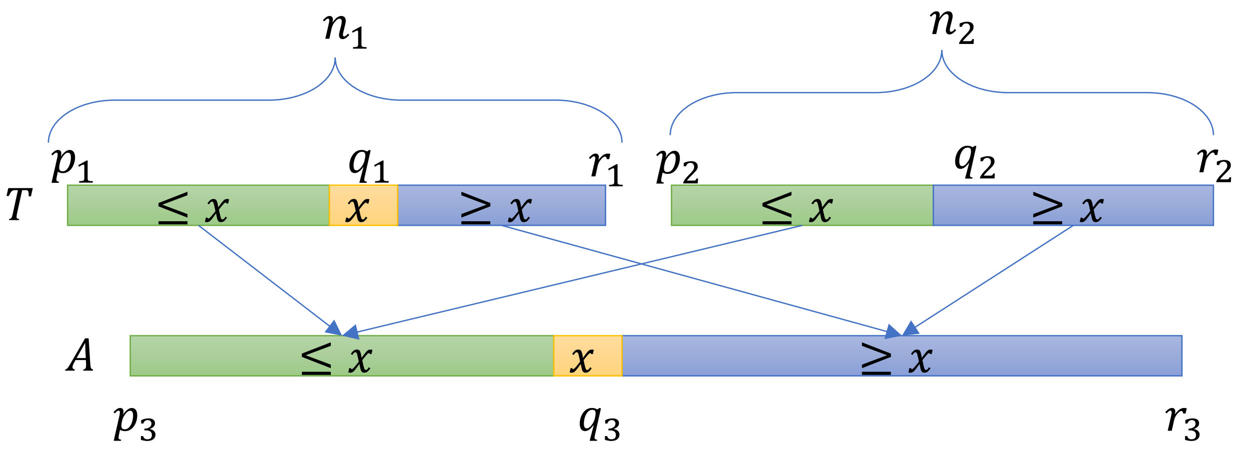
```
}
```

Merge sort

$n = n_1 + n_2$

Worst case number of elements?

```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```



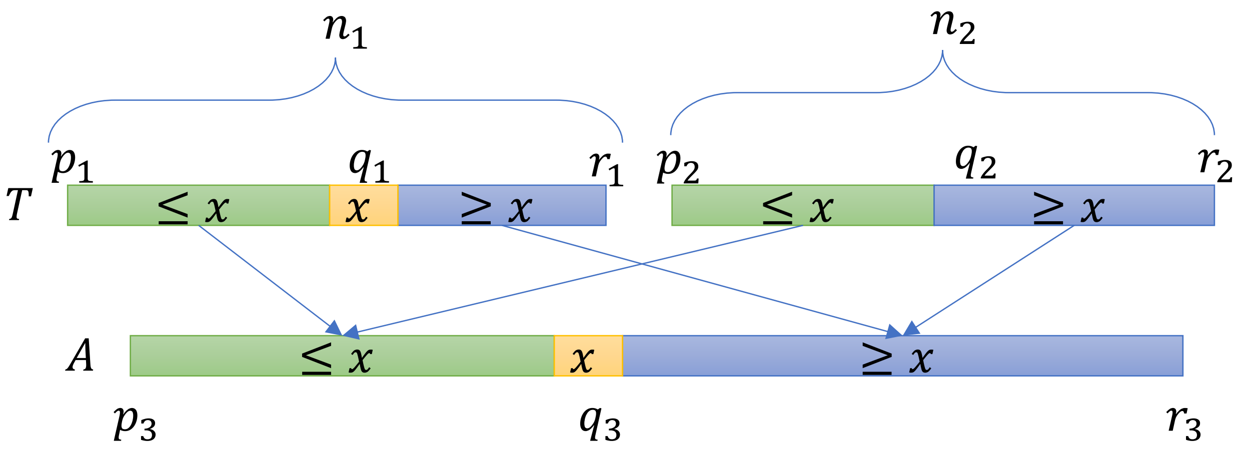
Merge sort

$n = n_1 + n_2$

Worst case number of elements?

- 3/4n

```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```

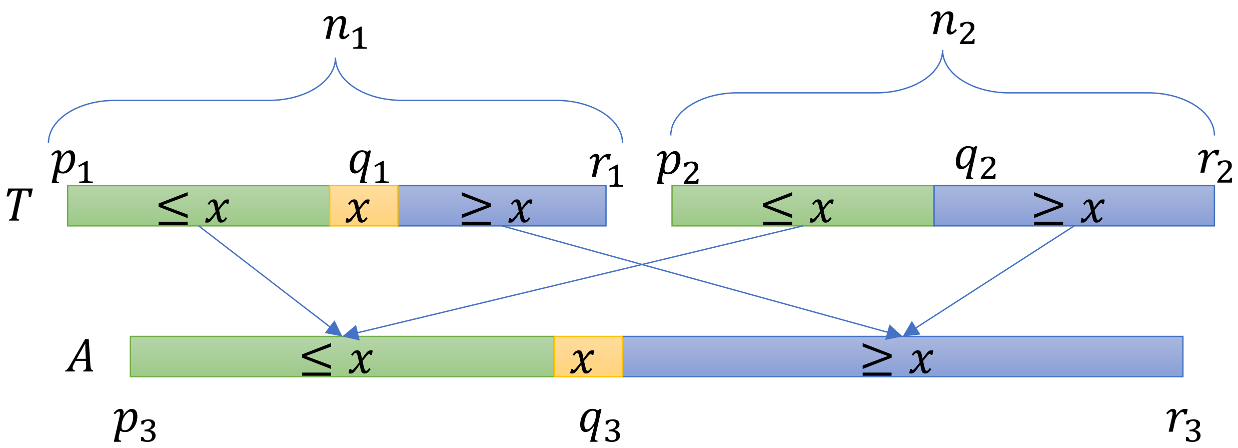


Merge sort

$n = n_1 + n_2$

Worst case number of elements?

- 3/4n ($n_2 \leq (n_1 + n_2)/2 = n/2$)
- $n_1/2 + n_2 \leq (n_1 + n_2)/2 + n_2/2 \leq 3n/4$

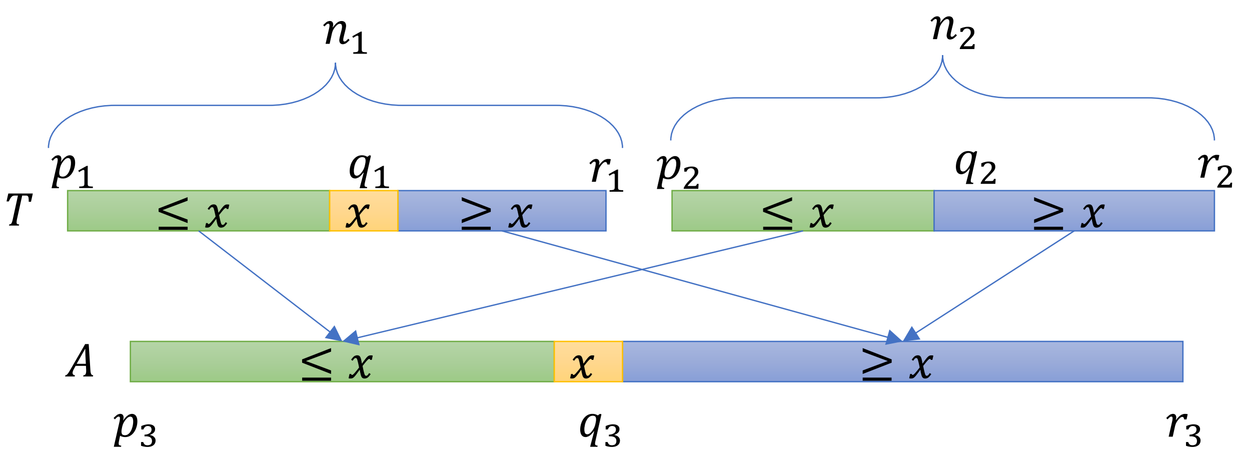



```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```

Merge sort

$$T_\infty(n) = T(3n/4) + \Theta(?)$$

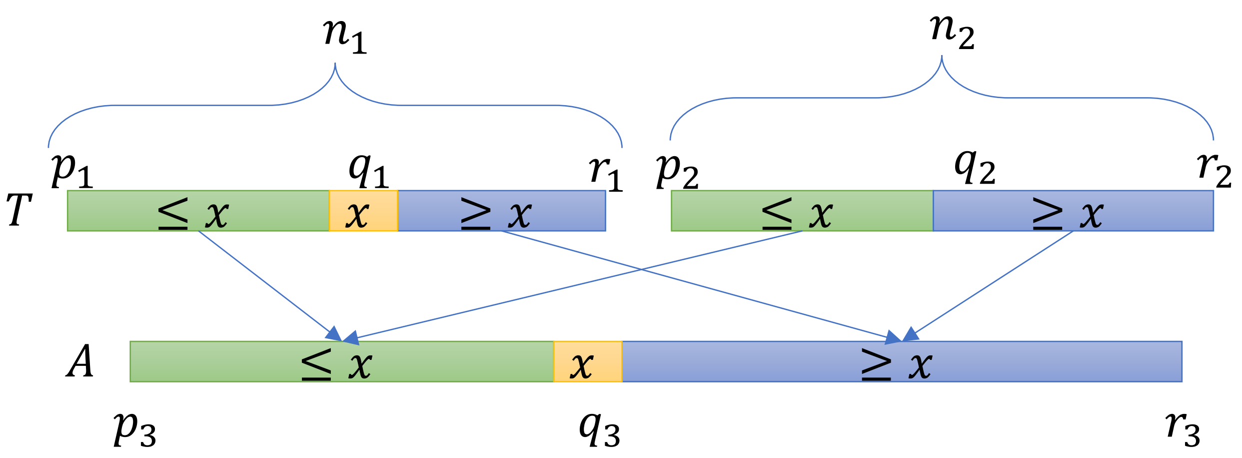
```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```



Merge sort

$$T_\infty(n) = T(3n/4) + \Theta(\log n)$$

```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```

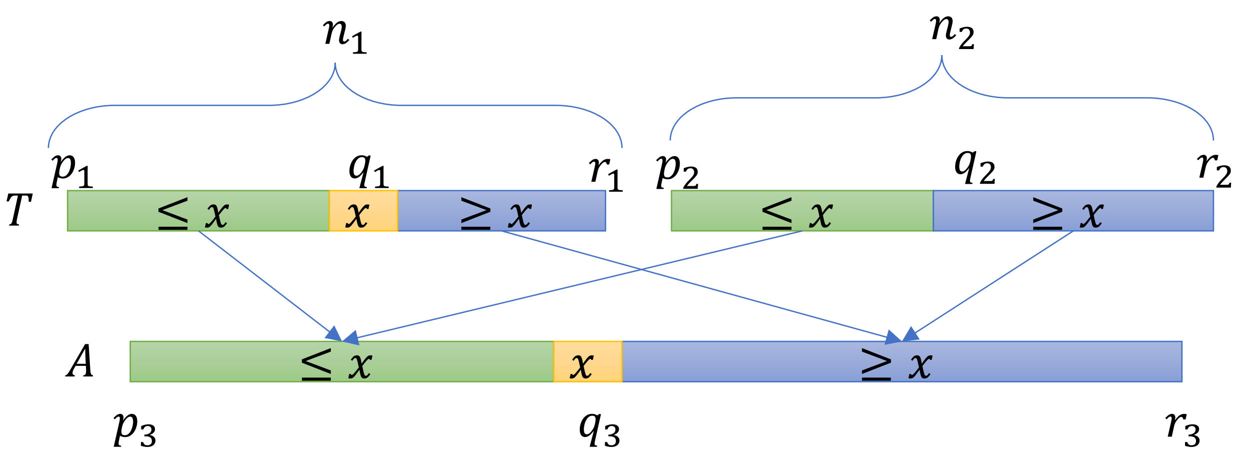


Merge sort

$$T_\infty(n) = T(3n/4) + \Theta(\log n)$$

- Cannot solve with master method
- $T_\infty(n) = \Theta((\log n)^2)$

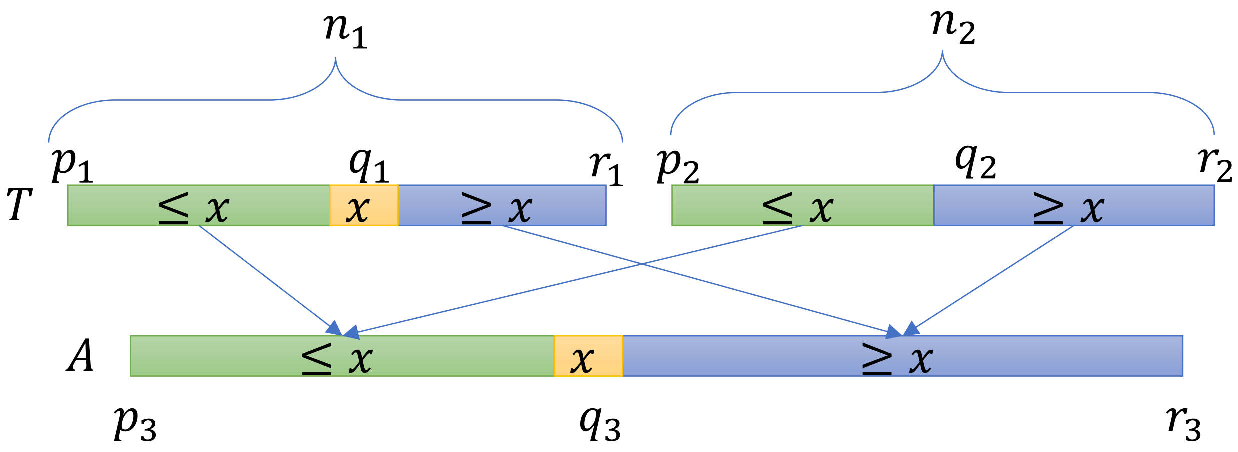
```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```



Work	
Span	$\Theta((\log n)^2)$
Parallelism	

Merge sort

$$T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(\log n), \frac{1}{4} < \alpha < \frac{3}{4}$$



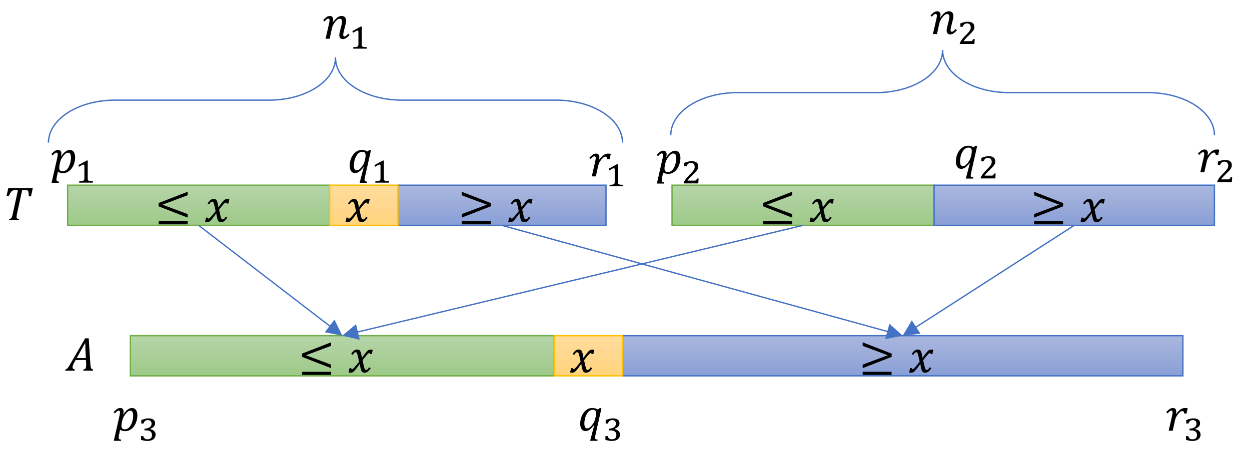

```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```

Work	
Span	$\Theta((\log n)^2)$
Parallelism	

Merge sort

$T(n) = \Theta(n)$

- Can be shown by substitution
  - $T(n) = c_1 n \; - c_2 \log n$
  - $T(n) = \mathrm{O}(\mathrm{n})$
  - $T(n) = \Omega(\mathrm{n})$ as each element is copied




```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```

Work	$\Theta(n)$
Span	$\Theta((\log n)^2)$
Parallelism	$\Theta(n/(\log n)^2)$

Merge sort

```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        spawn mergesort(a, p, q);
        mergesort(a, q+1, r);
        sync; pmerge(a, p, q, r);
    }
}
```

$T(n) =$

```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```

Work	
Span	
Parallelism	

Work	$\Theta(n)$
Span	$\Theta((\log n)^2)$
Parallelism	$\Theta(n/(\log n)^2)$

Merge sort

```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        spawn mergesort(a, p, q);
        mergesort(a, q+1, r);
        sync; pmerge(a, p, q, r);
    }
}
```

$$T(n) = 2T(n/2) + \Theta(n)$$

```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```

Work	
Span	
Parallelism	

Work	$\Theta(n)$
Span	$\Theta((\log n)^2)$
Parallelism	$\Theta(n/(\log n)^2)$

Merge sort

```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        spawn mergesort(a, p, q);
        mergesort(a, q+1, r);
        sync; pmerge(a, p, q, r);
    }
}
```

$$T(n) = 2T(n/2) + \Theta(n)$$

Work	$\Theta(n \log n)$
Span	
Parallelism	

```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```

Work	$\Theta(n)$
Span	$\Theta((\log n)^2)$
Parallelism	$\Theta(n/(\log n)^2)$

Merge sort

```
void mergesort(int* a, int p, int r) {
    if (p < r) {
        int q = (p+r)/2;
        spawn mergesort(a, p, q);
        mergesort(a, q+1, r);
        sync; pmerge(a, p, q, r);
    }
}
```

$$T_\infty(n) = T_\infty(n/2) + \Theta((\log n)^2)$$

```
void pmerge(...) {
    if (n1 < n2) swap ((p1,r1,n1), (p2,
r2,n2));
    int q1 = (p1+r1)/2;
    int q2 = bsearch();
    int q3 = p3+(q1-p1)+(q2-p2);
spawn pmerge(T,a,p1,q1-1,p2,q2-1,p3);
    pmerge(T,a,q1+1,r1,q2,r2,q3+1);
sync;
}
```

Work	$\Theta(n \log n)$
Span	$\Theta((\log n)^3)$
Parallelism	$\Theta(n/(\log n)^2)$

Work	$\Theta(n)$
Span	$\Theta((\log n)^2)$
Parallelism	$\Theta(n/(\log n)^2)$

BACKUP