Teaching Parallel Programming to Computer Science Freshmen

Leonardo B. A. Vasconcelos

Department of Computer Science

PUC Minas

Teaching Parallel Programming to Computer Science Freshmen

- This activity was planned to students of the course called Algorithms and Data Structures II of the undergraduate Computer Science curriculum at PUC Minas and it can be ported to any course related to OpenMP programming.

Outline

- Why parallel programming?
- About OpenMP
- “Hello world” in C
- Basic directives
  - *–omp parallel*
  - *–omp parallel for*
  - *–omp sections*
  - *–omp barrier*
  - *–omp master*
  - *–omp cirtical*
- Variable scope
- Reduction clause
- Speedup and efficiency
- How to parallelize a sort algorithm
- Selection Sort
- Merge Sort
- Shell Sort

Why parallel programming?

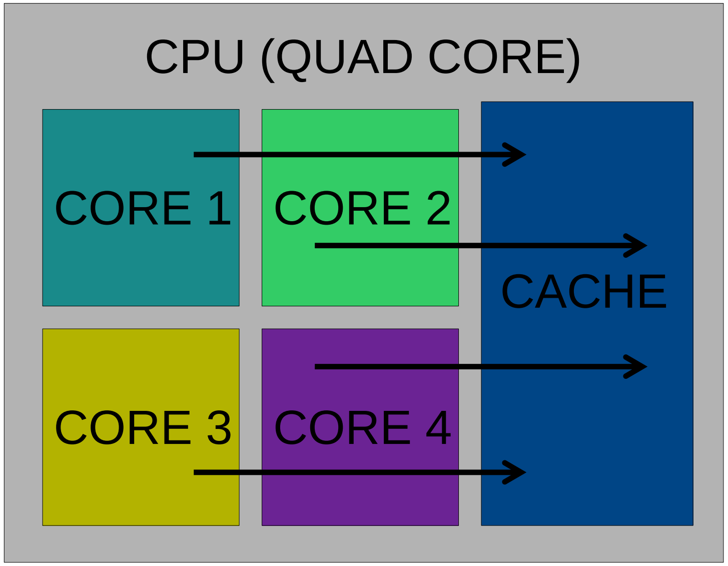
- Advantages
  - Parallel architectures available!
  - Performance improvement, low execution time.
- Disadvantages
  - The programmer needs to explict the parallelism
    - But, if you are a programmer, don’t worry!

About OpenMP

- OpenMP is an API to parallel programming based on shared memory.
- Languages available: C, C++ and Fortran.
- OpenMP is based on “fork and join” execution model.
- OpenMP is composed of:
  - Compiling directives
  - Function libraries
  - Environment variables

Parallel programming models

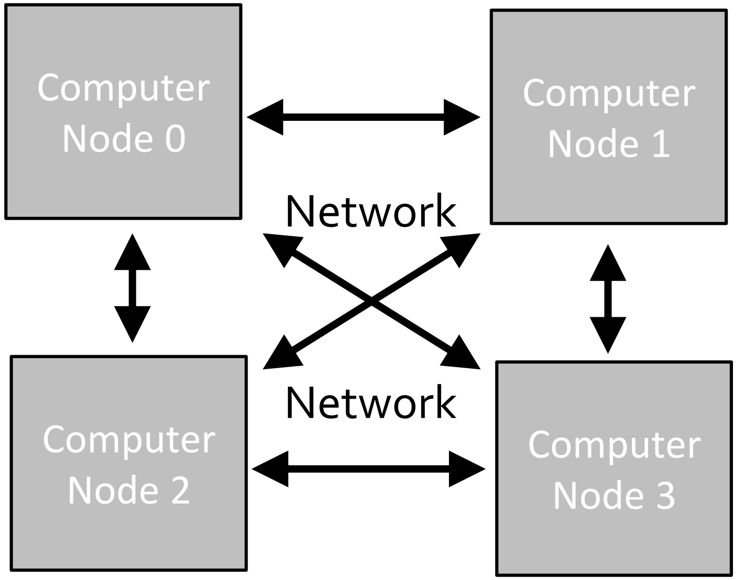
Shared variable



Each core can execute more than one thread from the same parallel application.The shared memory is used to exchange data between threads.

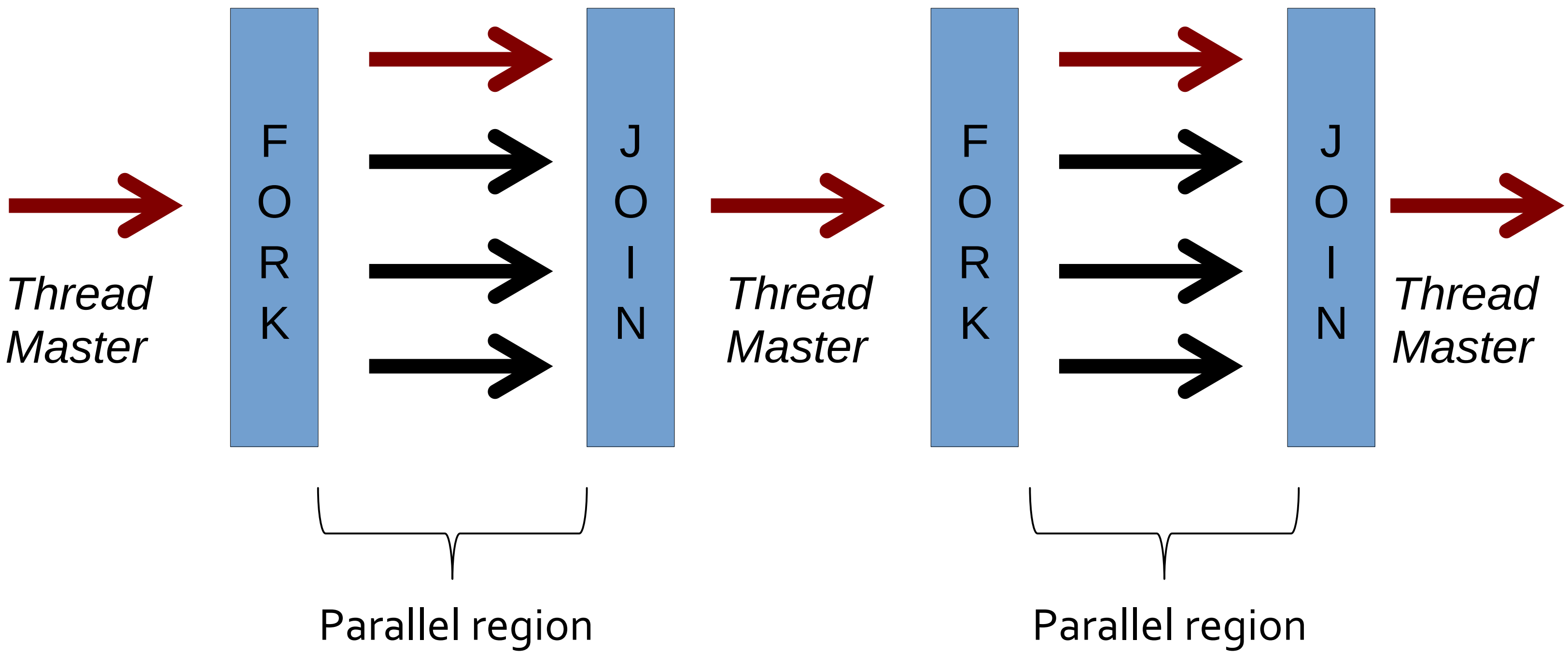
OpenMP is present here!

Message passing



Each machine executes a process from The same parallel application.
All communication is based on message passing through the network.

Fork and Join approach



“Hello world” in C

```
#include<stdio.h>

int main(){
        printf(“Hello world”);
        return 0;
}
```

Linux compiling with GCC
gcc helloworld.c -o helloworld

How to execute in Linux
./helloworld

```
#include<stdio.h>

int main(){
        int Number = 10;
        printf(“Number = %d”, Number);
        return 0;
}
```

“omp parallel” directive

```
#include<stdio.h>
#include<omp.h>

main(){
        printf(“Sequential %d\n”, omp_get_num_threads());
        #pragma omp parallel
        {
                printf(“Parallel %d\n”, omp_get_num_threads());
        }
        printf(“Sequential %d\n”, omp_get_num_threads());
}
```

GCC compiling with OpenMP:
gcc -fopenmp prog.c -o prog

“omp parallel” directive

```
#include<stdio.h>
#include<omp.h>

main(){
        printf(“Sequential %d\n”, omp_get_num_threads());
        #pragma omp parallel
        {
                printf(“Parallel %d\n”, omp_get_num_threads());
        }
        printf(“Sequential %d\n”, omp_get_num_threads());
}
```

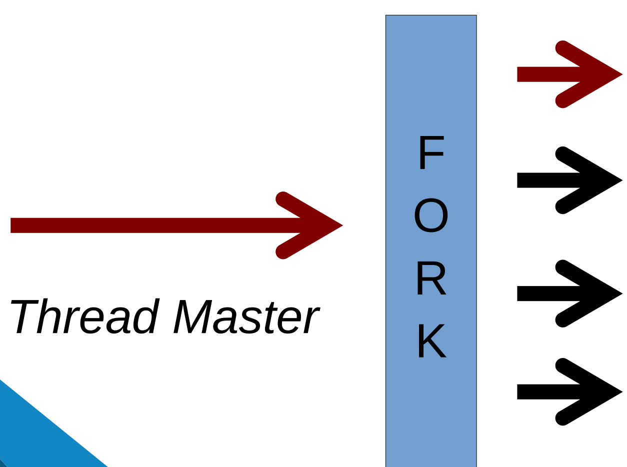
```
Sequential 1
```

Thread Master

“omp parallel” directive

```
#include<stdio.h>
#include<omp.h>

main(){
        printf("Sequential %d\n", omp_get_num_threads());
        #pragma omp parallel
        {
                printf("Parallel %d\n", omp_get_num_threads());
        }
        printf("Sequential %d\n", omp_get_num_threads());
}
```

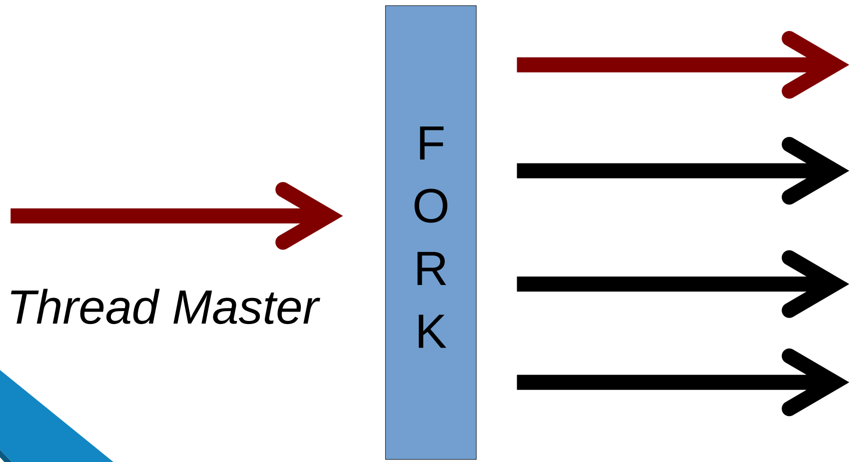
```
Sequential 1
```



“omp parallel” directive

```
#include<stdio.h>
#include<omp.h>

main(){
        printf(“Sequential %d\n”, omp_get_num_threads());
        #pragma omp parallel
        {
                printf(“Parallel %d\n”, omp_get_num_threads());
        }
        printf(“Sequential %d\n”, omp_get_num_threads());
}
```

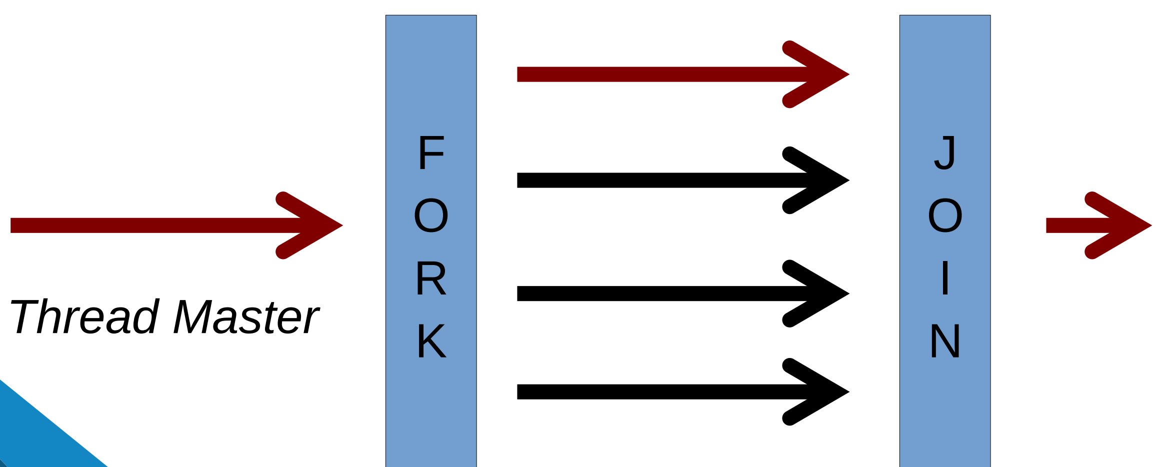
```
Sequential 1
Parallel 4
Parallel 4
Parallel 4
Parallel 4
```



“omp parallel” directive

```
#include<stdio.h>
#include<omp.h>

main(){
        printf(“Sequential %d\n”, omp_get_num_threads());
        #pragma omp parallel
        {
                printf(“Parallel %d\n”, omp_get_num_threads());
        }
        printf(“Sequential %d\n”, omp_get_num_threads());
}
```

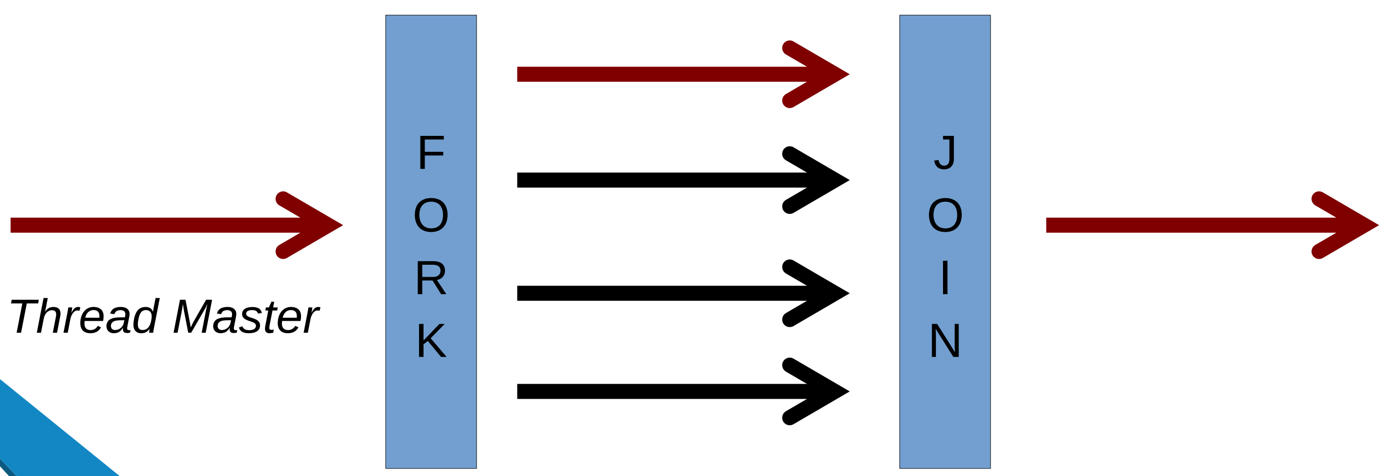
```
Sequential 1
Parallel 4
Parallel 4
Parallel 4
Parallel 4
```



“omp parallel” directive

```
#include<stdio.h>
#include<omp.h>

main(){
        printf("Sequential %d\n", omp_get_num_threads());
        #pragma omp parallel
        {
                printf("Parallel %d\n", omp_get_num_threads());
        }
        printf("Sequential %d\n", omp_get_num_threads());
}
```

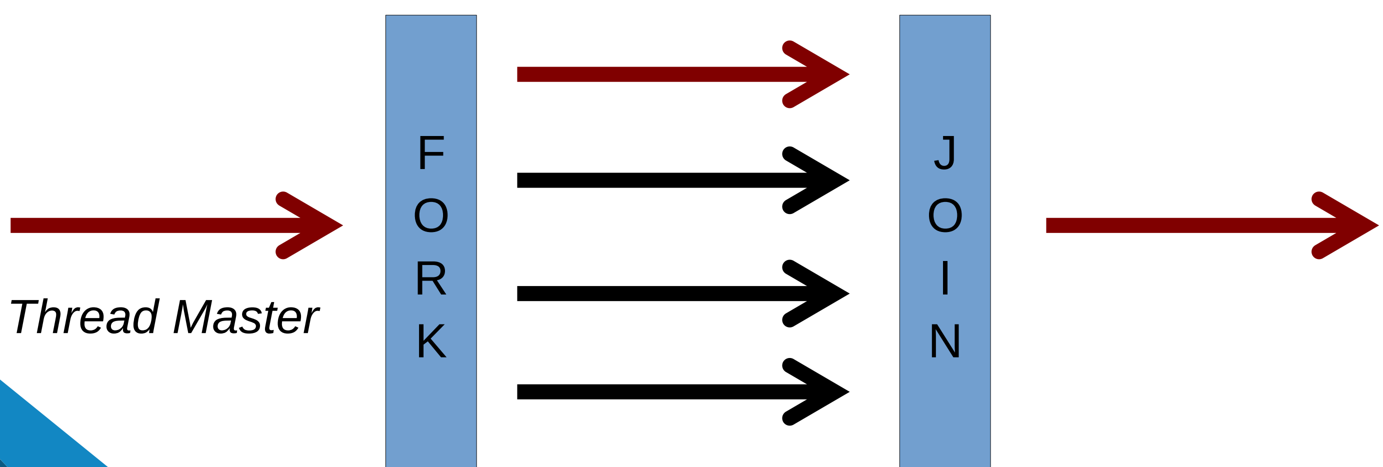
```
Sequential 1
Parallel 4
Parallel 4
Parallel 4
Parallel 4
Sequential 1
```



“omp parallel” directive

```
#include<stdio.h>
#include<omp.h>

main(){
        printf(“Sequential %d\n”, omp_get_num_threads());
        #pragma omp parallel
        {
                printf(“Thread %d\n”, omp_get_thread_num());
        }
        printf(“Sequential %d\n”, omp_get_num_threads());
}
```

```
Sequential 1
Thread 2
Thread 0
Thread 3
Thread 1
Sequential 1
```



“omp parallel for” directive

```
#include<stdio.h>
#include<omp.h>

main(){
        int i;
        for(i = 0; i < 9; i=i+1)
                printf(“I: %d \n”, i);
}
```

```
I: 0
I: 1
I: 2
I: 3
I: 4
I: 5
I: 6
I: 7
I: 8
```

0	1	2	3	4	5	6	7	8

“omp parallel for” directive

```
#include<stdio.h>
#include<omp.h>

main(){
        int i;
        #pragma omp parallel for num_threads(3)
        for(i = 0; i < 9; i=i+1)
                printf(“ID:%d, I: %d \n”, omp_get_thread_num(), i);
}
```

```
ID: 0 I: 0
ID: 2 I: 6
ID: 0 I: 1
ID: 1 I: 3
ID: 1 I: 4
ID: 1 I: 5
ID: 2 I: 7
ID: 2 I: 8
ID: 0 I: 2
```

Review the directives

Serial

```
int main(){
int i;
 for(i = 0; i < 3; i=i+1)
                printf("I = %d\n", i);
}
```

#pragma omp parallel

```
int main(){
int i;
 #pragma omp parallel
 for(i = 0; i < 3; i=i+1)
                printf("I = %d\n", i);
}
```

#pragma omp parallel for

```
int main(){
int i;
 #pragma omp parallel for
 for(i = 0; i < 3; i=i+1)
                printf("I = %d\n", i);
}
```

Review the directives

Serial

```
int main(){
int i;
 for(i = 0; i < 3; i=i+1)
		printf("I = %d\n", i);
}
```

```
I = 0
```

#pragma omp parallel

```
int main(){
int i;
 #pragma omp parallel
 for(i = 0; i < 3; i=i+1)
		printf("I = %d\n", i);
}
```

```
I = 0
I = 0
I = 0
```

#pragma omp parallel for

```
int main(){
int i;
 #pragma omp parallel for
 for(i = 0; i < 3; i=i+1)
		printf("I = %d\n", i);
}
```

```
I = 0
I = 1
I = 2
```

Review the directives

Serial

```
int main(){
int i;
 for(i = 0; i < 3; i=i+1)
		printf("I = %d\n", i);
}
```

```
I = 0
I = 1
```

#pragma omp parallel

```
int main(){
int i;
 #pragma omp parallel
 for(i = 0; i < 3; i=i+1)
		printf("I = %d\n", i);
}
```

```
I = 0
I = 0
I = 0
I = 1
I = 1
I = 1
```

#pragma omp parallel for

```
int main(){
int i;
 #pragma omp parallel for
 for(i = 0; i < 3; i=i+1)
		printf("I = %d\n", i);
}
```

```
I = 0
I = 1
I = 2
```

Review the directives

Serial

```
int main(){
int i;
 for(i = 0; i < 3; i=i+1)
		printf("I = %d\n", i);
}
```

```
I = 0
I = 1
I = 2
```

#pragma omp parallel

```
int main(){
int i;
 #pragma omp parallel
 for(i = 0; i < 3; i=i+1)
		printf("I = %d\n", i);
}
```

```
I = 0
I = 0
I = 0
I = 1
I = 1
I = 1
I = 2
I = 2
I = 2
```

#pragma omp parallel for

```
int main(){
int i;
 #pragma omp parallel for
 for(i = 0; i < 3; i=i+1)
		printf("I = %d\n", i);
}
```

```
I = 0
I = 1
I = 2
```

Review the directives

The final result could be different at least for one of them?

Serial	#pragma omp parallel	#pragma omp parallel for
I = 0 I = 1 I = 2	I = 0 I = 0 I = 0 I = 1 I = 1 I = 1 I = 2 I = 2 I = 2	I = 0 I = 1 I = 2

“omp parallel sections” directive

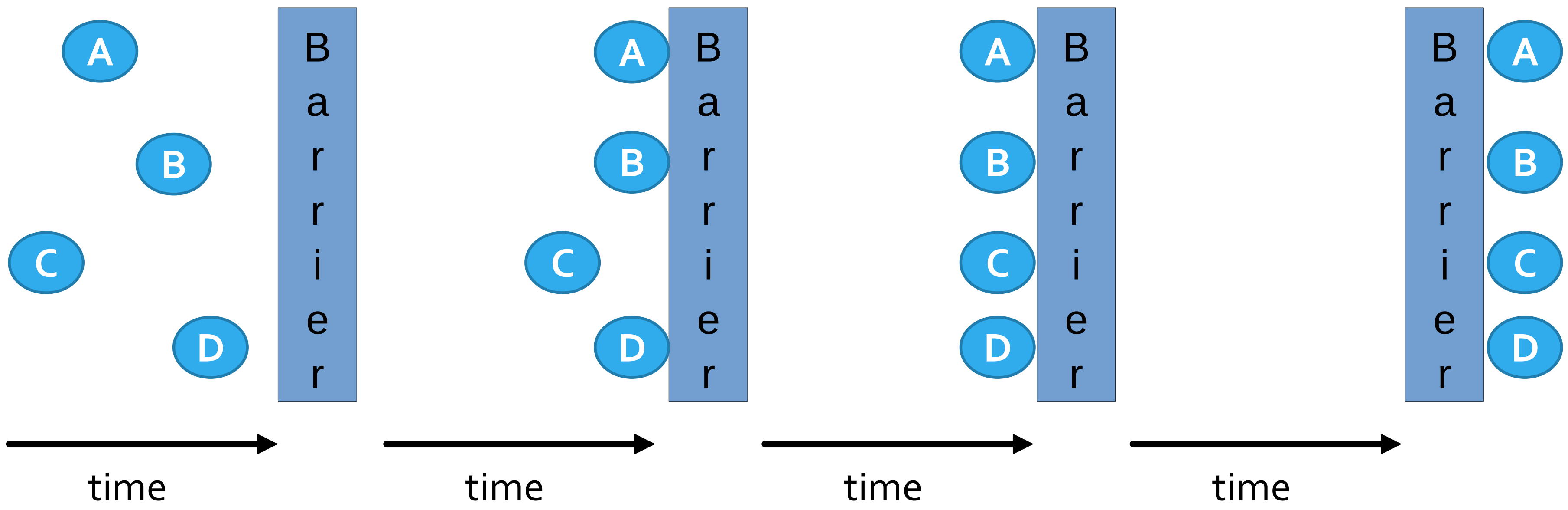
```c
#include<stdio.h>
#include<omp.h>

main(){
        #pragma omp parallel sections
        {
          #pragma omp section
                processA();
          #pragma omp section
                processB();
          #pragma omp section
                processC();
         }
}
```

6 *Threads*	2 *Threads*
Thread 4	*Thread* 0
Thread 2	*Thread* 1
Thread 0	*Thread* 1

- With 6 available cores, 3 threads will be executed, each one for each section.
- With 2 threads available, one of them will execute two sections.

“omp barrier” directive

Threads A, B, C and D



Synchronization: All threads must achieve a barrier before to continue the execution.

Note: Try to avoid the barrier, since it adds communication overhead (synchronization).

“omp barrier” directive

```
main(){
        omp_set_thread_num(3);
        #pragma omp parallel
        {
                printf(“Phase 1”);
                printf(“Phase 2”);
        }
}
```

```
Phase 1
Phase 2
Phase 1
Phase 2
Phase 1
Phase 2
```

```
main(){
        omp_set_thread_num(3);
        #pragma omp parallel
        {
                printf(“Phase 1”);
                #pragma omp barrier
                printf(“Phase 2”);
        }
}
```

```
Phase 1
Phase 1
Phase 1
Phase 2
Phase 2
Phase 2
```

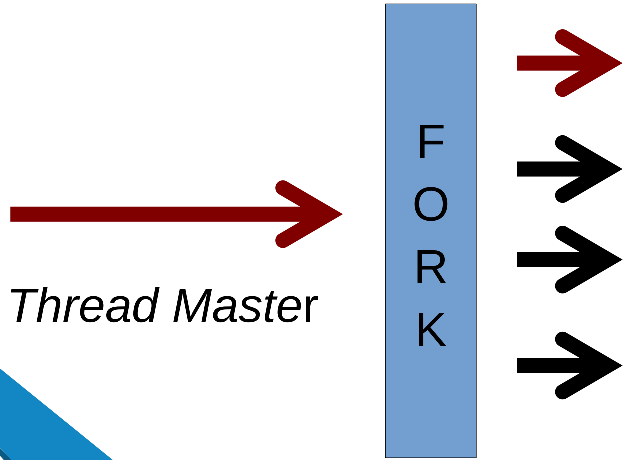
“omp master” directive

```
main(){
        #pragma omp parallel
        {
                printf(“Thread %d\n”, omp_get_thread_num());
                #pragma omp barrier
                #pragma omp master
                {
                  printf(“Master %d\n”, omp_get_thread_num());
                }
        }
}
```

- *Thread master* ID is always 0.

“omp master” directive

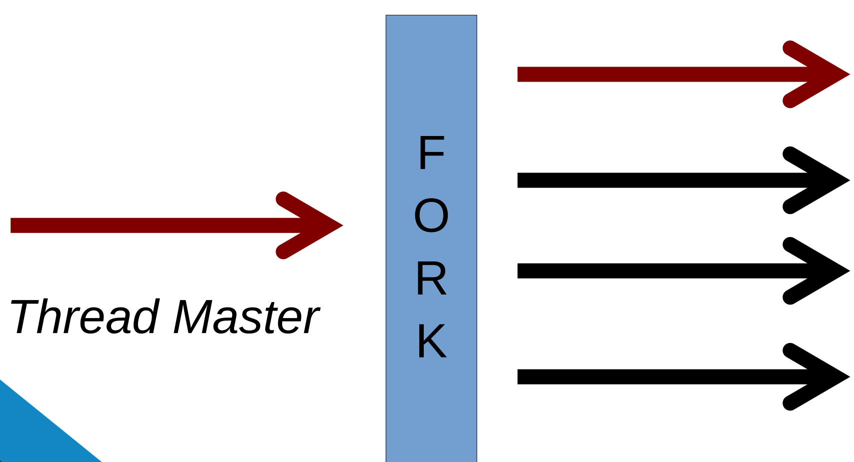
```
main(){
        #pragma omp parallel
        {
                printf(“Thread %d\n”, omp_get_thread_num());
                #pragma omp barrier
                #pragma omp master
                {
                  printf(“Master %d\n”, omp_get_thread_num());
                }
        }
}
```



“omp master” directive

```
main(){
        #pragma omp parallel
        {
                printf(“Thread %d\n”, omp_get_thread_num());
                #pragma omp barrier
                #pragma omp master
                {
                  printf(“Master %d\n”, omp_get_thread_num());
                }
        }
}
```

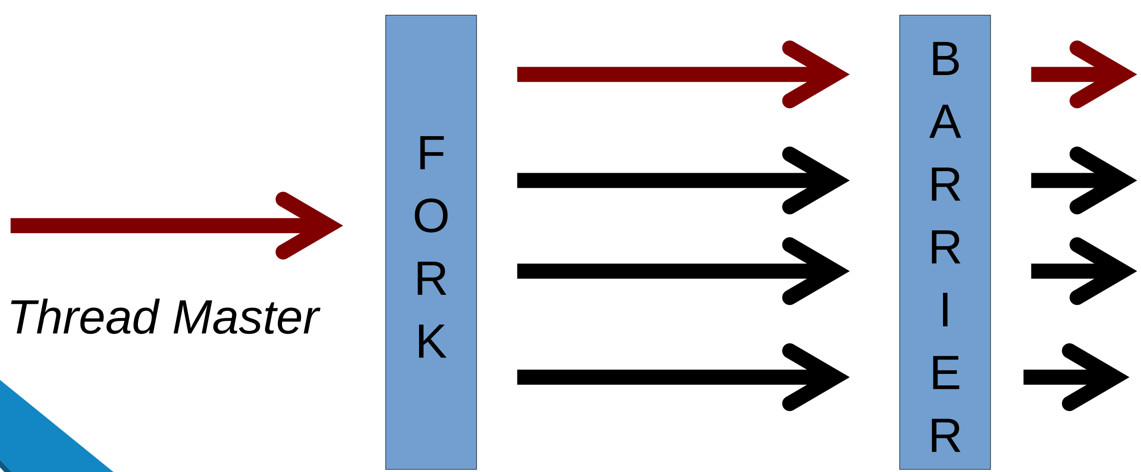
```
Thread 1
Thread 0
Thread 3
Thread 2
```



“omp master” directive

```
main(){
        #pragma omp parallel
        {
                printf(“Thread %d\n”, omp_get_thread_num());
                #pragma omp barrier
                #pragma omp master
                {
                  printf(“Master %d\n”, omp_get_thread_num());
                }
        }
}
```

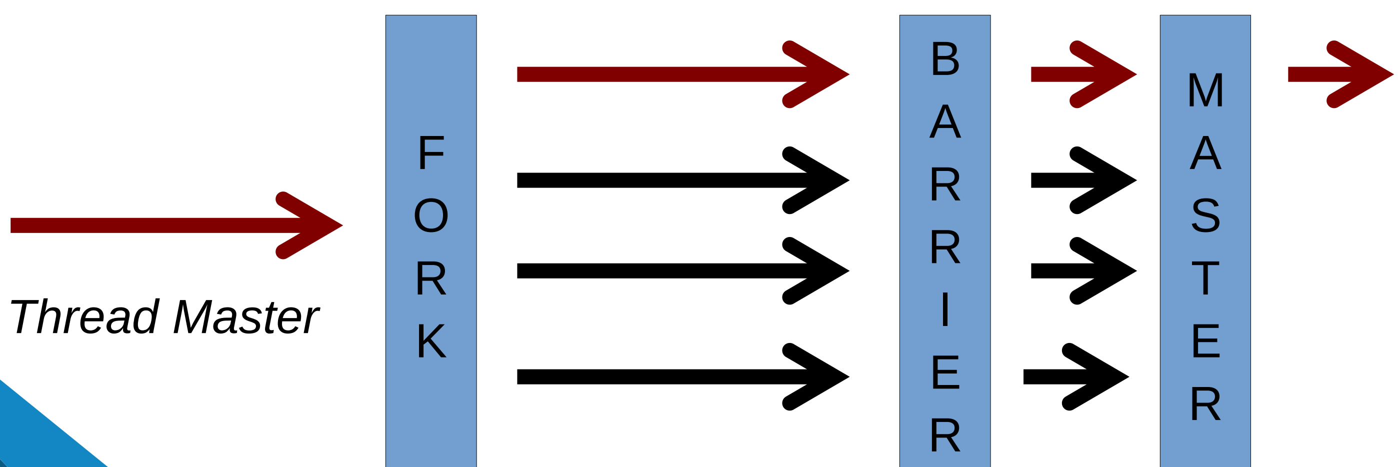
```
Thread 1
Thread 0
Thread 3
Thread 2
```



“omp master” directive

```
main(){
        #pragma omp parallel
        {
                printf(“Thread %d\n”, omp_get_thread_num());
                #pragma omp barrier
                #pragma omp master
                {
                  printf(“Master %d\n”, omp_get_thread_num());
                }
        }
}
```

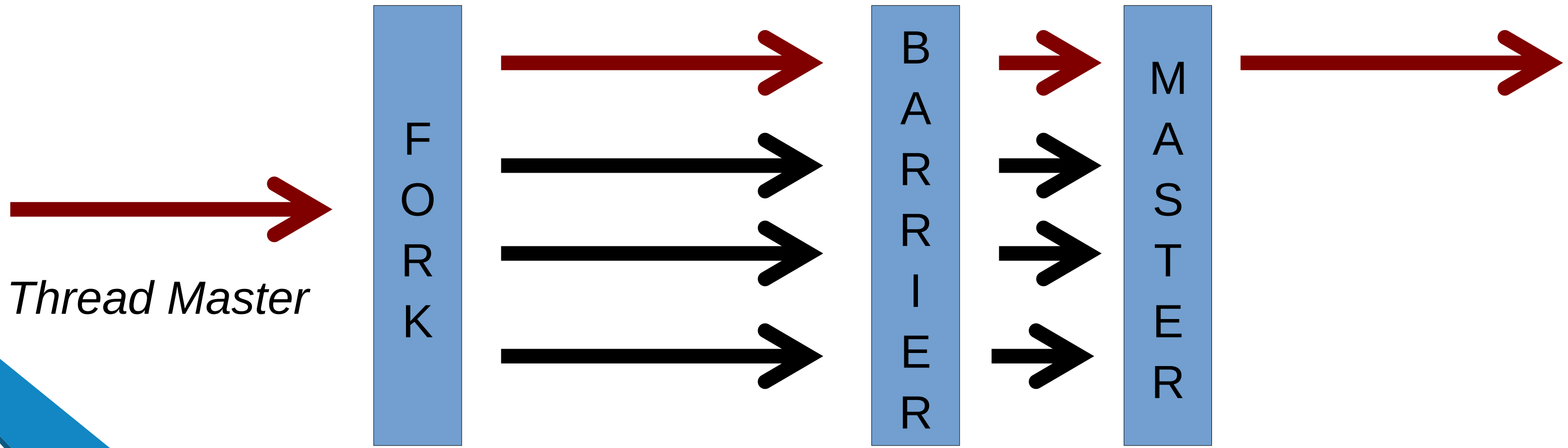
```
Thread 1
Thread 0
Thread 3
Thread 2
```



“omp master” directive

```
main(){
        #pragma omp parallel
        {
                printf("Thread %d\n", omp_get_thread_num());
                #pragma omp barrier
                #pragma omp master
                {
                  printf("Master %d\n", omp_get_thread_num());
                }
        }
}
```

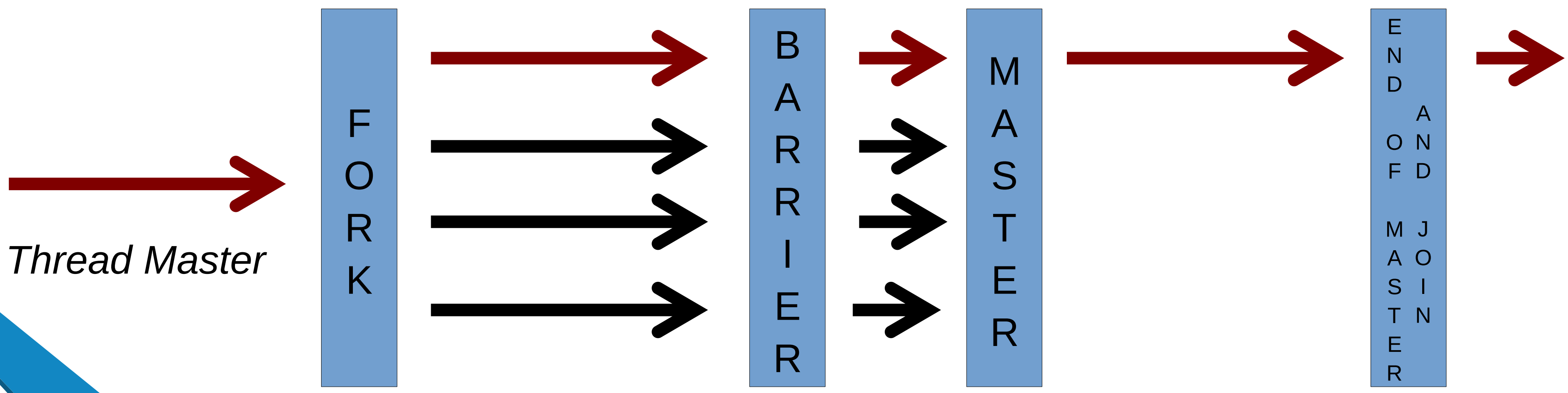
```
Thread 1
Thread 0
Thread 3
Thread 2
Master 0
```



“omp master” directive

```
main(){
        #pragma omp parallel
        {
                printf(“Thread %d\n”, omp_get_thread_num());
                #pragma omp barrier
                #pragma omp master
                {
                  printf(“Master %d\n”, omp_get_thread_num());
                }
        }
}
```

```
Thread 1
Thread 0
Thread 3
Thread 2
Master 0
```



“omp critical” directive

```
main() {
	int max = 0;
	#pragma omp parallel
	{
		int local = rand();
		#pragma omp barrier
		if(max < local)
			max = local;
	}
}
```

max	local - T0	local - T1	local - T2
N/A	N/A	N/A	N/A

T0, T1, T2

T0

T1

T2

This code generates a random number for each thread and the bigger will be stored in the **max** variable.

There is a race condition. **max** is a shared variable and all threads will write on it.

“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                if(max < local)
                        max = local;
        }
}
```

max	local - T0	local - T1	local - T2
0	N/A	N/A	N/A

T0, T1, T2

T0

T1

T2

Thread Master

“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                if(max < local)
                        max = local;
        }
}
```

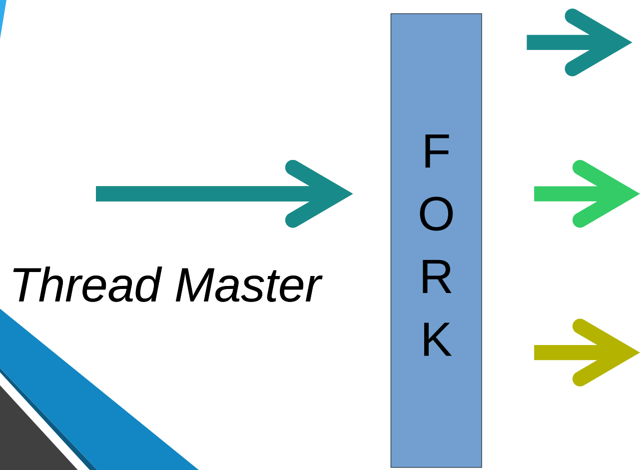
max	local - T0	local - T1	local - T2
0	N/A	N/A	N/A

T0, T1, T2

T0

T1

T2



“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                if(max < local)
                        max = local;
        }
}
```

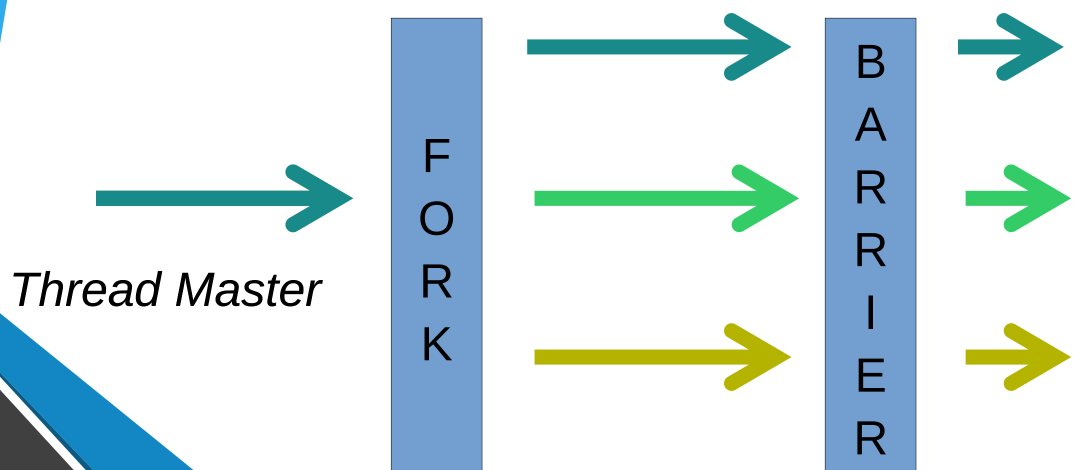
max	local - T0	local - T1	local - T2
0	5	3	7

T0, T1, T2

T0

T1

T2



“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                if(max < local)
                        max = local;
        }
}
```

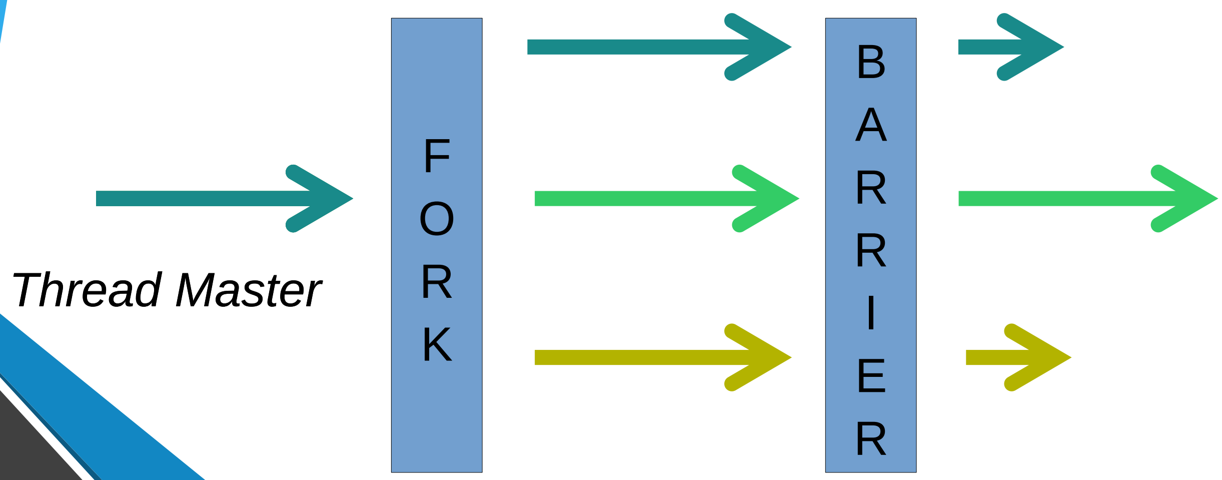
max	local - T0	local - T1	local - T2
3	5	3	7

T0, T1, T2

T0

T1

T2



“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                if(max < local)
                        max = local;
        }
}
```

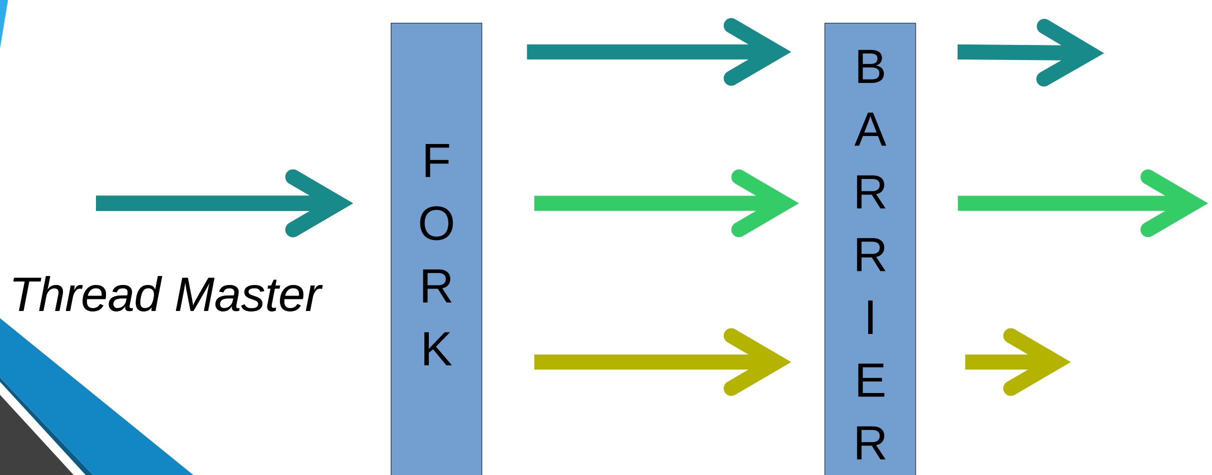
max	local - T0	local - T1	local - T2
3	5	3	7

T0, T1, T2

T0

T1

T2



“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                if(max < local)
                        max = local;
        }
}
```

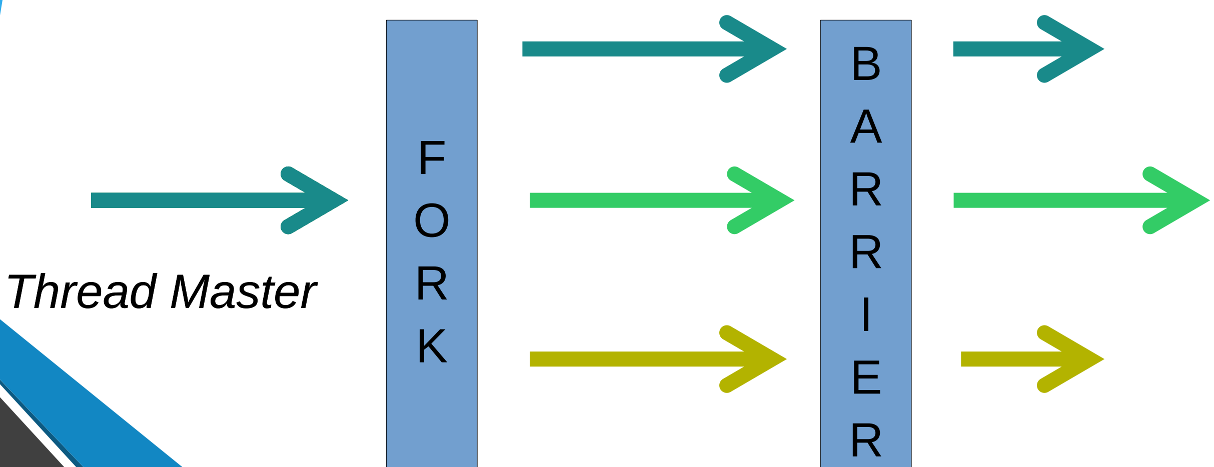
max	local - T0	local - T1	local - T2
3	5	3	7

T0, T1, T2

T0

T1

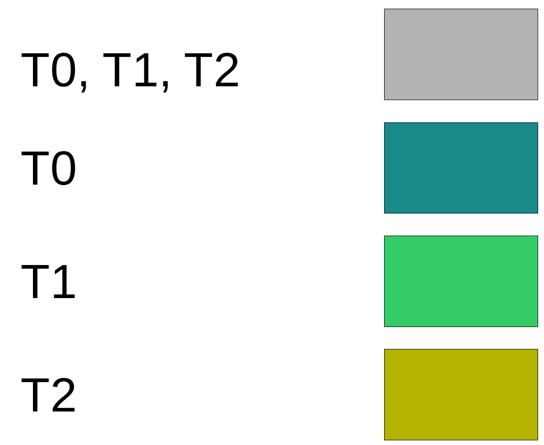
T2

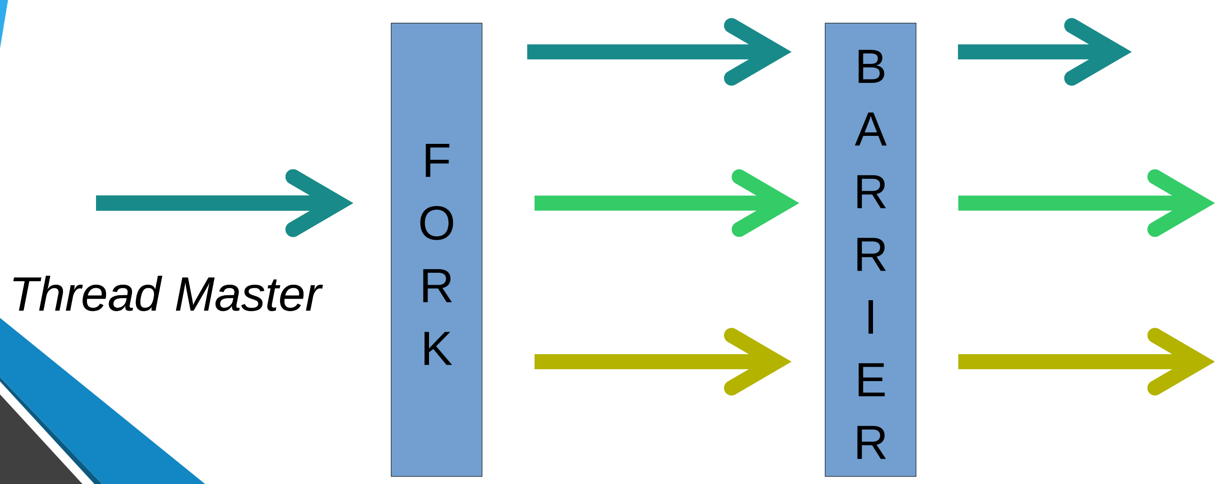


“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                if(max < local)
                        max = local;
        }
}
```

max	local - T0	local - T1	local - T2
7	5	3	7





“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                if(max < local)
                        max = local;
        }
}
```

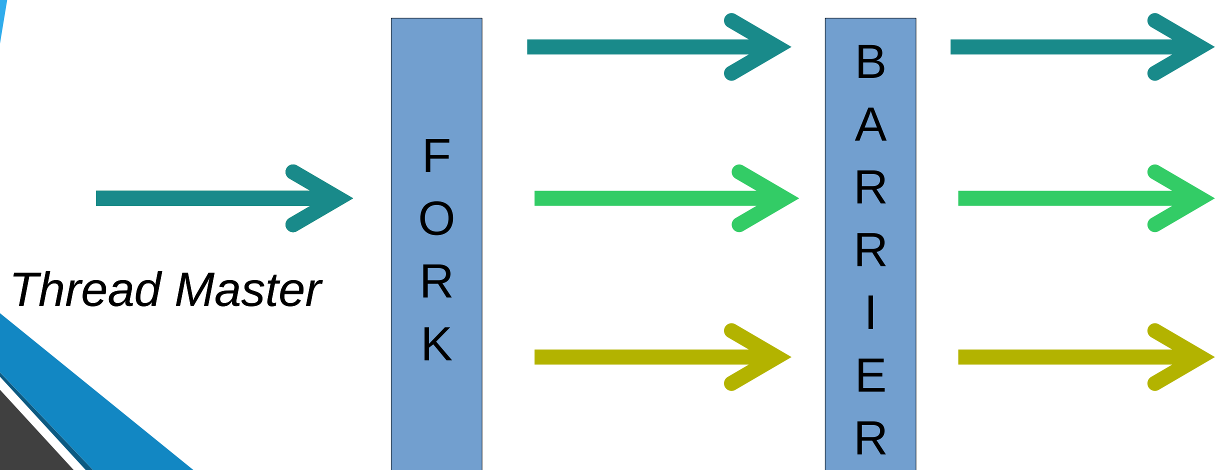
max	local - T0	local - T1	local - T2
5	5	3	7

T0, T1, T2

T0

T1

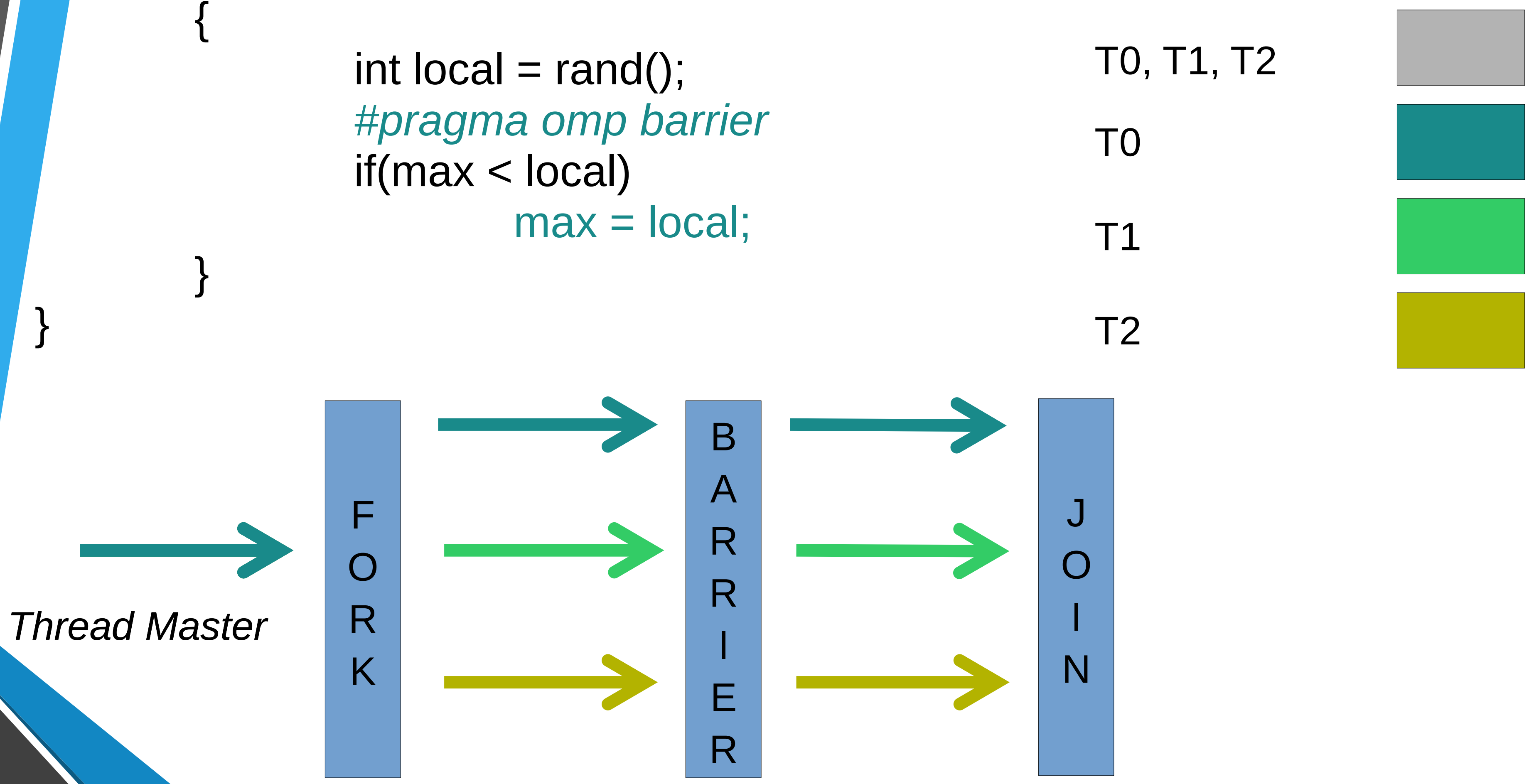
T2



“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                if(max < local)
                        max = local;
        }
}
```

max	local - T0	local - T1	local - T2
5	5	3	7



“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                if(max < local)
                        max = local;
        }
}
```

max	local - T0	local - T1	local - T2
5	5	3	7

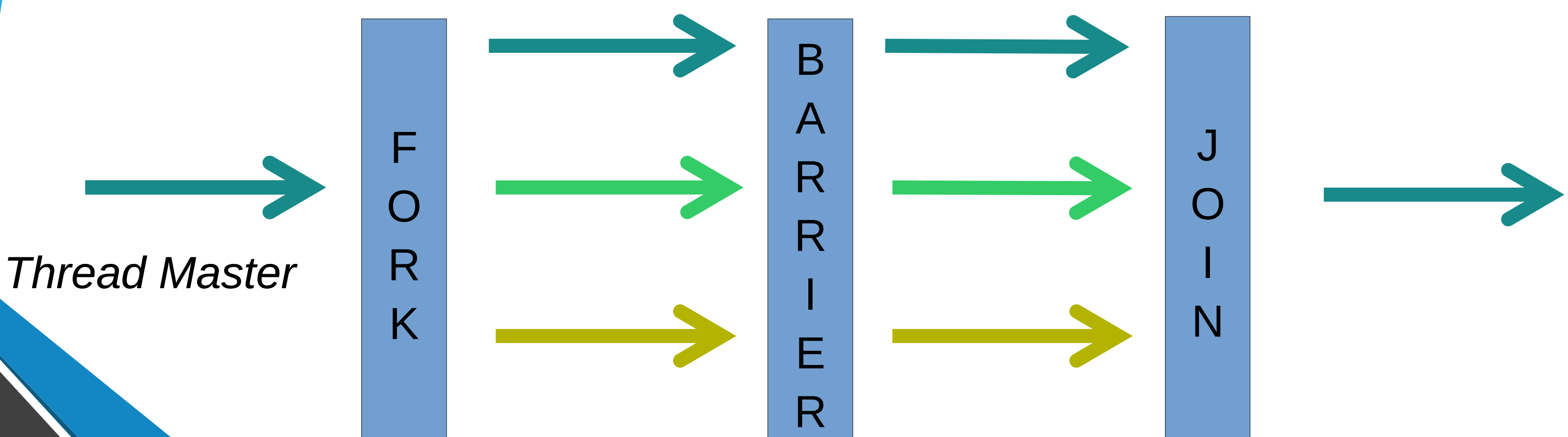
Resultado incorreto

T0, T1, T2

T0

T1

T2



“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                #pragma omp critical
                {
                        if(max < local)
                                max = local;
                }
        }
}
```

max	local - T0	local - T1	local - T2
N/A	N/A	N/A	N/A

T0, T1, T2

T0

T1

T2

“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                #pragma omp critical
                {
                        if(max < local)
                                max = local;
                }
        }
}
```

max	local - T0	local - T1	local - T2
0	N/A	N/A	N/A

T0, T1, T2

T0

T1

T2

Thread Master

“omp critical” directive

```
main() {
    int max = 0;
    #pragma omp parallel
    {
        int local = rand();
        #pragma omp barrier
        #pragma omp critical
        {
            if(max < local)
                max = local;
        }
    }
}
```

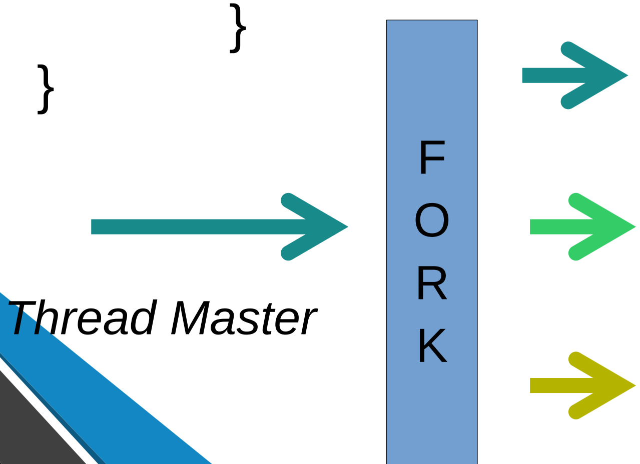
max	local - T0	local - T1	local - T2
0	N/A	N/A	N/A

T0, T1, T2

T0

T1

T2



"omp critical" directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                #pragma omp critical
                {
                        if(max < local)
                                max = local;
                }
        }
}
```

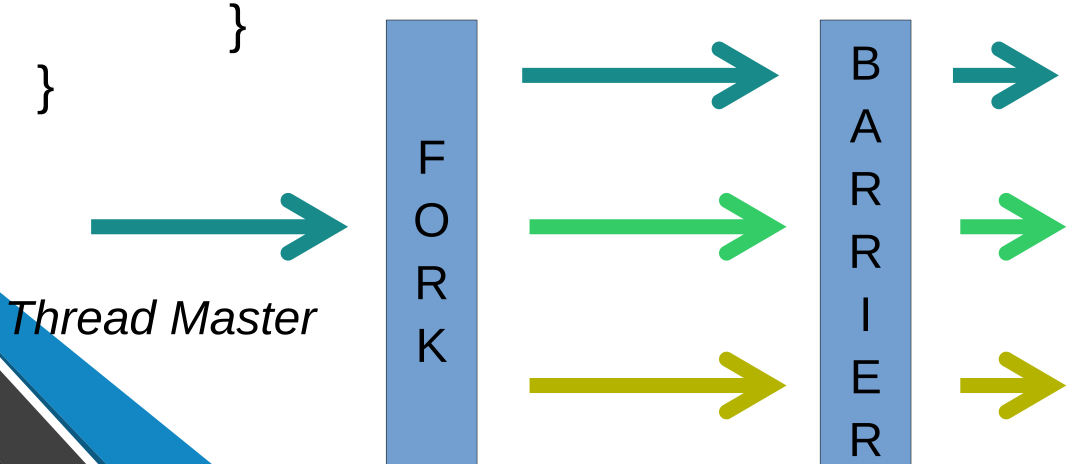
max	local - T0	local - T1	local - T2
0	5	3	7

T0, T1, T2

T0

T1

T2



“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                #pragma omp critical
                {
                        if(max < local)
                                max = local;
                }
        }
}
```

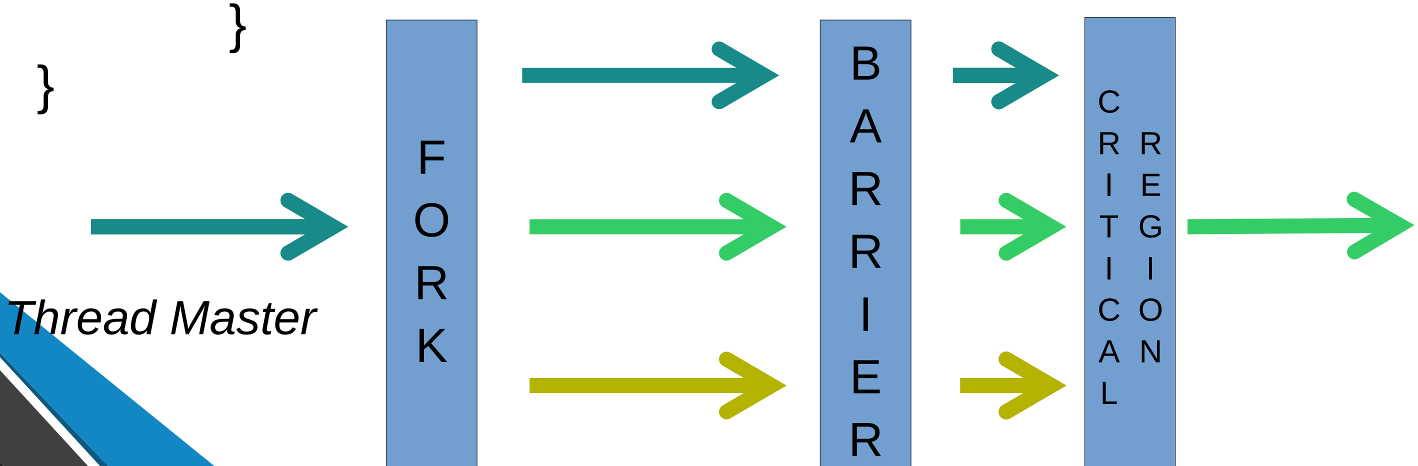
max	local - T0	local - T1	local - T2
3	5	3	7

T0, T1, T2

T0

T1

T2



"omp critical" directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                #pragma omp critical
                {
                        if(max < local)
                                max = local;
                }
        }
}
```

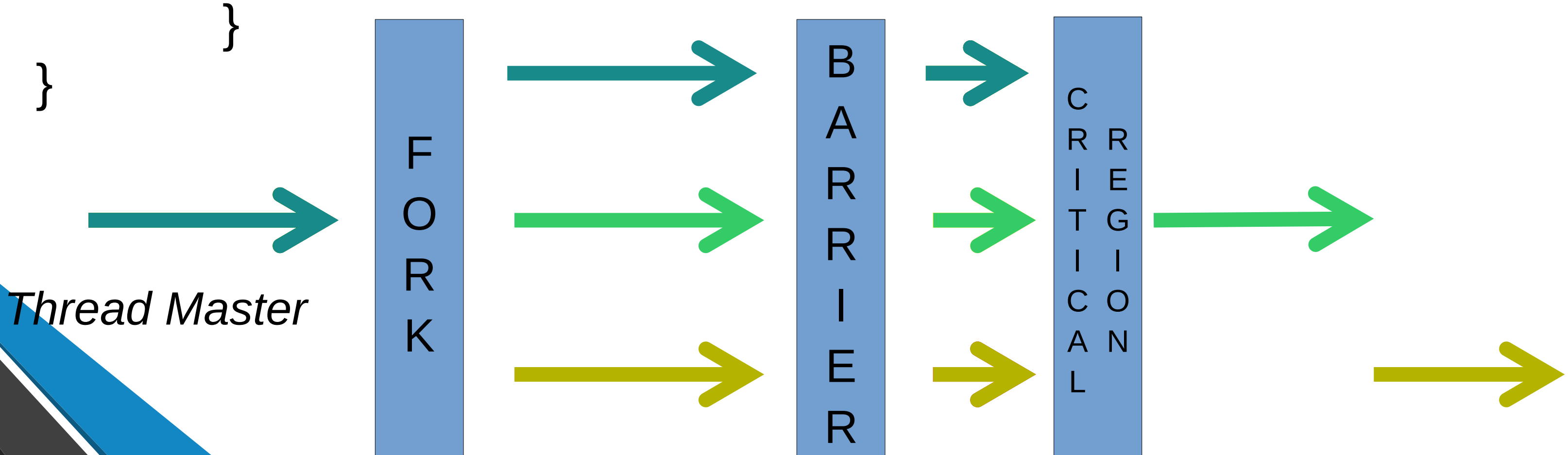
max	local - T0	local - T1	local - T2
7	5	3	7

T0, T1, T2

T0

T1

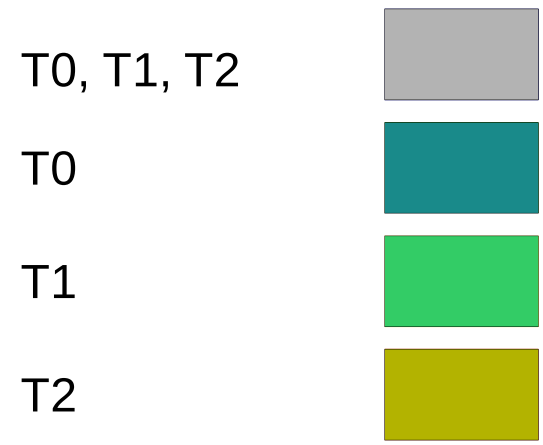
T2

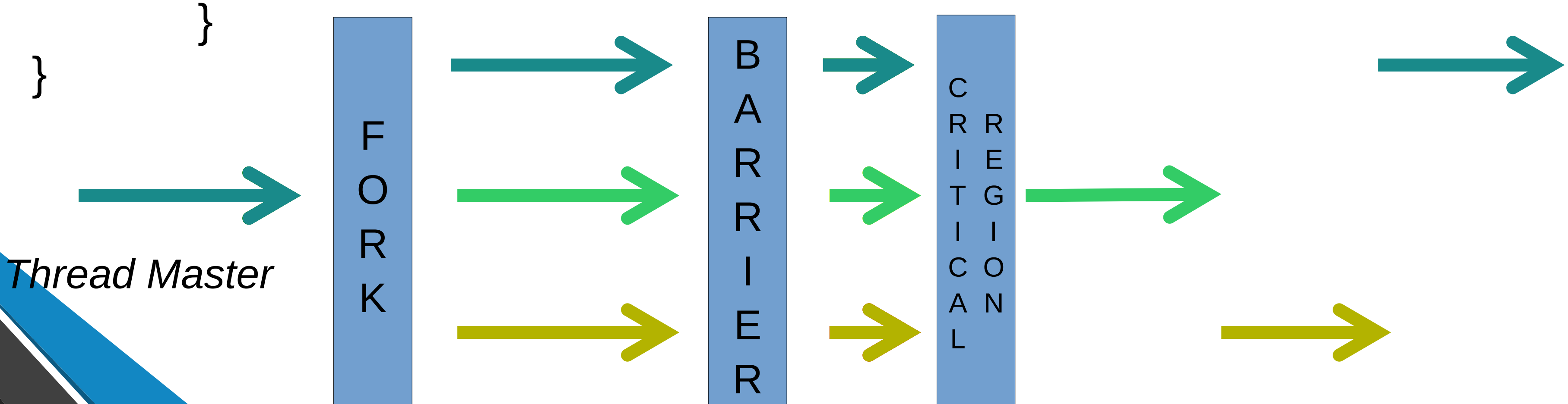


“omp critical” directive

```
main() {
        int max = 0;
        #pragma omp parallel
        {
                int local = rand();
                #pragma omp barrier
                #pragma omp critical
                {
                        if(max < local)
                                max = local;
                }
        }
}
```

max	local - T0	local - T1	local - T2
7	5	3	7

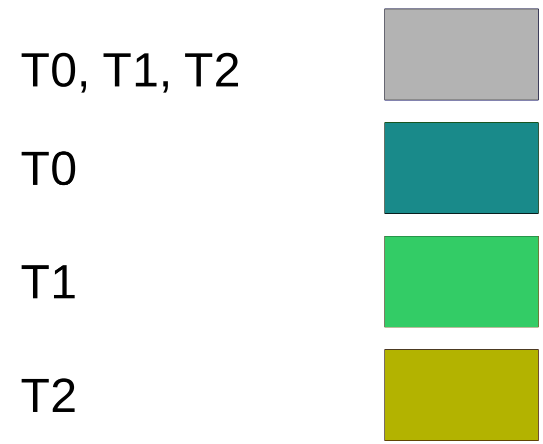


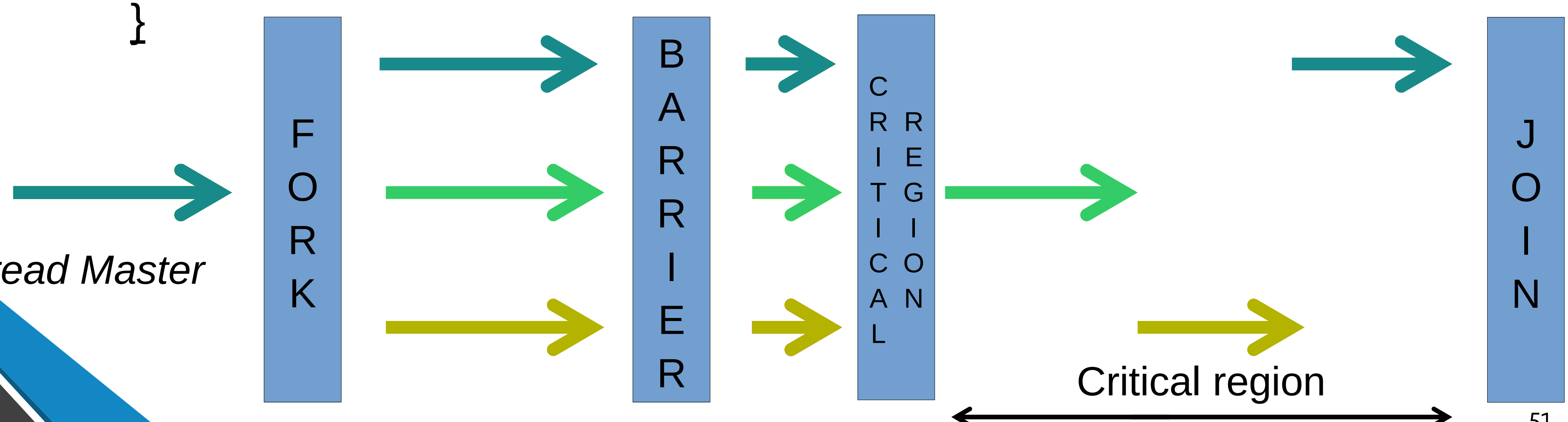


“omp critical” directive

```
main() {
    int max = 0;
    #pragma omp parallel
    {
        int local = rand();
        #pragma omp barrier
        #pragma omp critical
        {
            if(max < local)
                max = local;
        }
    }
}
```

max	local - T0	local - T1	local - T2
7	5	3	7





Variable scope

```
main() {
        int x, y;
        #pragma omp parallel
        {
                int z;
        }
}
```

X: shared
Y: shared
Z: private

```
main() {
        int x, y;
        #pragma omp parallel private (x)
        {
                int z;
        }
}
```

X: private
Y: shared
Z: private

```
main() {
        int x, y;
        #pragma omp parallel for
        for(y=0; y < 5; y++)
        {
                int z;
        }
}
```

X: shared
Y: private
Z: private

Reduction clause

Serial version

```
int i, sum = 0;
for(i = 0; i < N; i++){
        sum+= rand();
}
```

Reduction clause

Serial version

```
int i, sum = 0;
for(i = 0; i < N; i++){
        sum+= rand();
}
```

Parallel version

```
int i, sum = 0;
#pragma omp parallel for
for(i = 0; i < N; i++){
        sum+= rand();
}
```

Reduction clause

Serial version

```
int i, sum = 0;
for(i = 0; i < N; i++){
        sum+= rand();
}
```

Parallel version

```
int i, sum = 0;
#pragma omp parallel for
for(i = 0; i < N; i++){
        sum+= rand();
}
```

Reduction clause

Serial version

```
int i, sum = 0;
for(i = 0; i < N; i++){
        sum+= rand();
}
```

Parallel version

```
int i, sum = 0;
#pragma omp parallel for
for(i = 0; i < N; i++){
        sum+= rand();
}
```

Parallel version using Reduction

```
int i, sum = 0;
#pragma omp parallel for reduction (+:sum)
for(i = 0; i < N; i++){
        sum+= i;
}
```

Granularity

Definition: It is related to the task size a thread receives to execute.

Fine-grained granularity: When this task is relatively short.
In this case, the application scalability can be weakened.

Coarse-grained granularity: When this task is relatively long.
In this case, the application scalability can be improved.

Note: The granularity is depended on the number of threads, since we need to divide the overall workload into tasks to each thread. So, if the number of threads is bigger the task size is smaller to each thread.

Speedup and Efficiency

- Speedup is a performance increase after an application parallelization. To calculate it, just divide the serial time (st) by parallel time (pt). The ideal result is a number equal to the number of cores used to run the parallel version.

- Example: st = 10 seconds, pt = 5 seconds.

  4 cores used to run the parallel version
  Ideal speedup = 4
  Real speedup = st/pt = 10/5 = 2.

Speedup and Efficiency

- Efficiency is a metric derived from speedup.

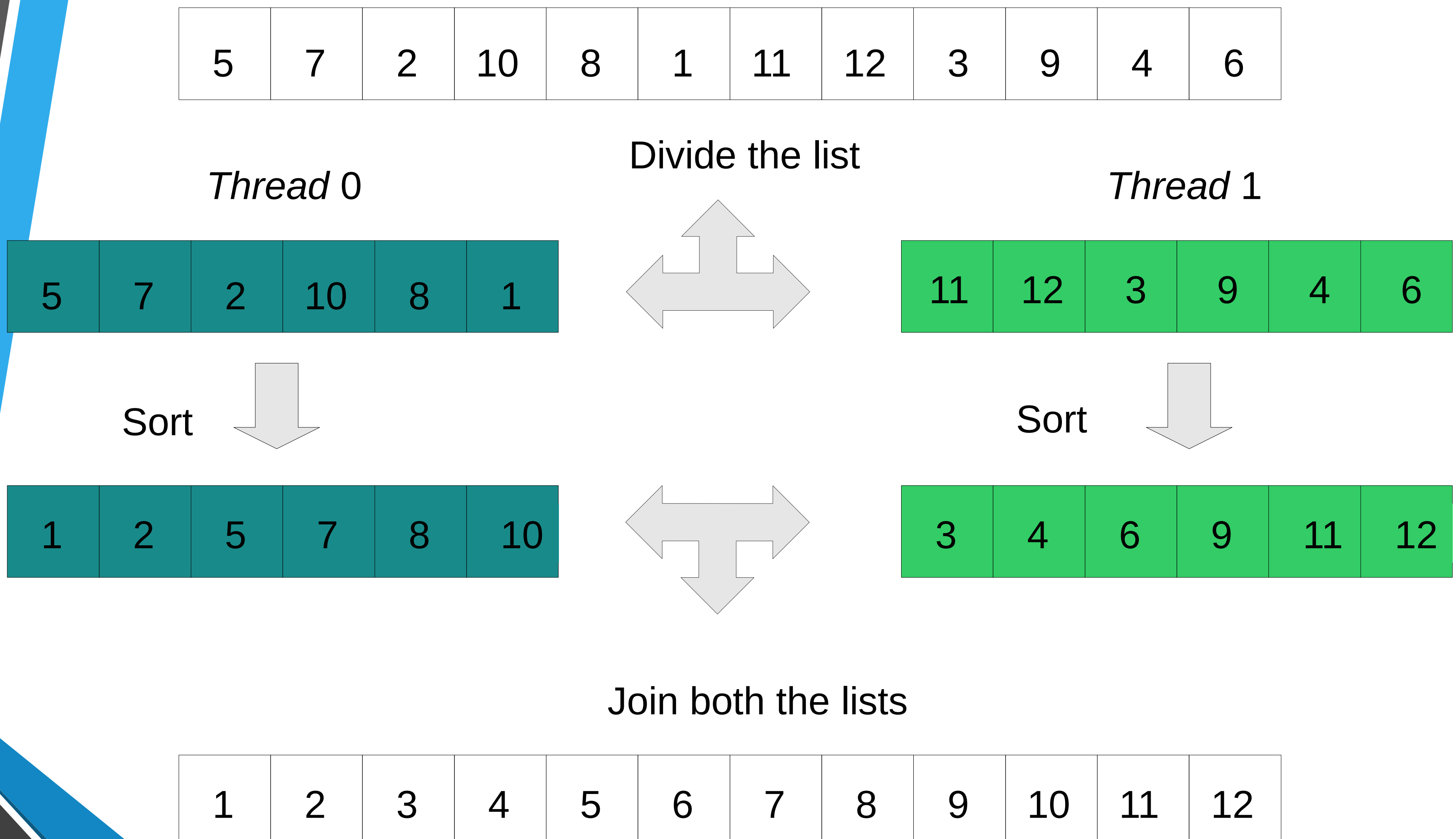
  Efficiency = Speedup/Number of cores

  The efficiency varies from 0 to 1, where 1 means the maximum improvement, in order words, 100%.

- Example: Speedup = 2, Number of cores = 4
  2/4 = 0,5 (50% of efficiency).

  Note: There are situations when the efficiency is higher than 1. In that cases, there are influence of cache memories, etc.

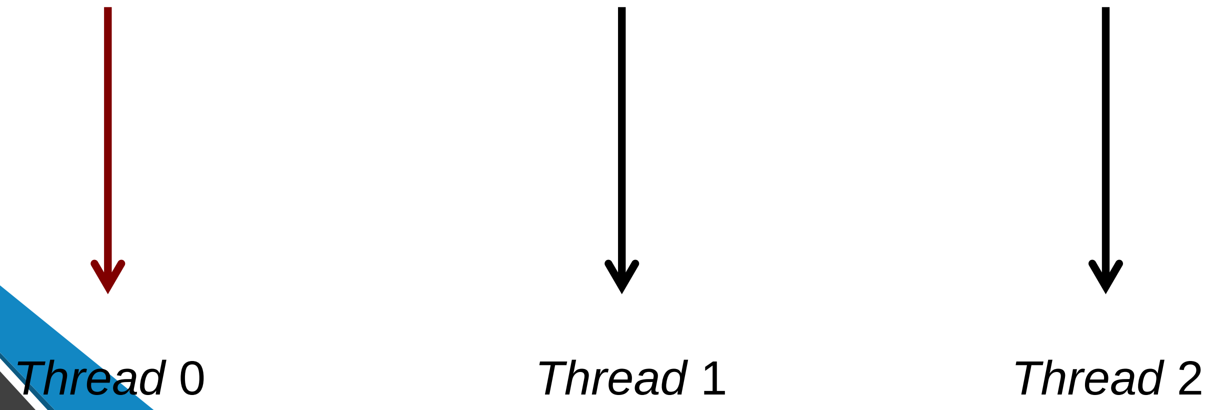
A Parallel Sorting Algorithm



A Parallel Sorting Algorithm

5	7	2	10	8	1	11	12	3	9	4	6

Global size: 12



A Parallel Sorting Algorithm

5	7	2	10	8	1	11	12	3	9	4	6

Global size = 12

Thread 0 *Thread* 1 *Thread* 2

$$\text{Local size} = \frac{\text{Global size}}{\text{Number of Threads}}$$

Start = ID * Local size
Final = Start + Local size -1

A Parallel Sorting Algorithm

5	7	2	10	8	1	11	12	3	9	4	6

Global size = 12

Start = 0 * 4 = 0
Final = 0 + 4 -1 = 3

Start = 1 * 4 = 4
Final = 4 + 4 - 1 = 7

Start = 2 * 4 = 8
Final = 8 + 4 -1 = 11

Local size = 12/3 = 4
Start = ID*4
Final = Start + Local size -1

Thread 0

Thread 1

Thread 2

A Parallel Sorting Algorithm

5	7	2	10	8	1	11	12	3	9	4	6

Global size = 12

Start = 0 * 4 = 0
Final = 0 + 4 -1 = 3

Start = 1 * 4 = 4
Final = 4 + 4 - 1 = 7

Start = 2 * 4 = 8
Final = 8 + 4 -1 = 11

Local size = 12/3 = 4
Start = ID*4
Final = Start + Local size -1

Thread 0

Thread 1

Thread 2

A Parallel Sorting Algorithm

Partitioning the array among threads

```
…
#pragma omp parallel
{
        int start, local_size;
        local_size = global_size / omp_get_num_threads();
        start = omp_get_thread_num() * local_size;
        …
}
```

A Parallel Sorting Algorithm

Sort lists

```
…
#pragma omp parallel
{
        int start, local_size;
        local_size = global_size / omp_get_num_thread();
        start = omp_get_thread_num() * local_size;
        bubbleSort(arr+start, local_size);
        ...
}
```

A Parallel Sorting Algorithm

Sort lists

```
…
#pragma omp parallel
{
        int start, local_size;
        local_size = global_size / omp_get_num_thread();
        start = omp_get_thread_num() * local_size;
        insertionSort(arr+start, local_size);
        ...
}
```

A Parallel Sorting Algorithm

Sort lists

```
…
#pragma omp parallel
{
        int start, local_size;
        local_size = global_size / omp_get_num_thread();
        start = omp_get_thread_num() * local_size;
        selectionSort(arr+start, local_size);
        ...
}
```

A Parallel Sorting Algorithm

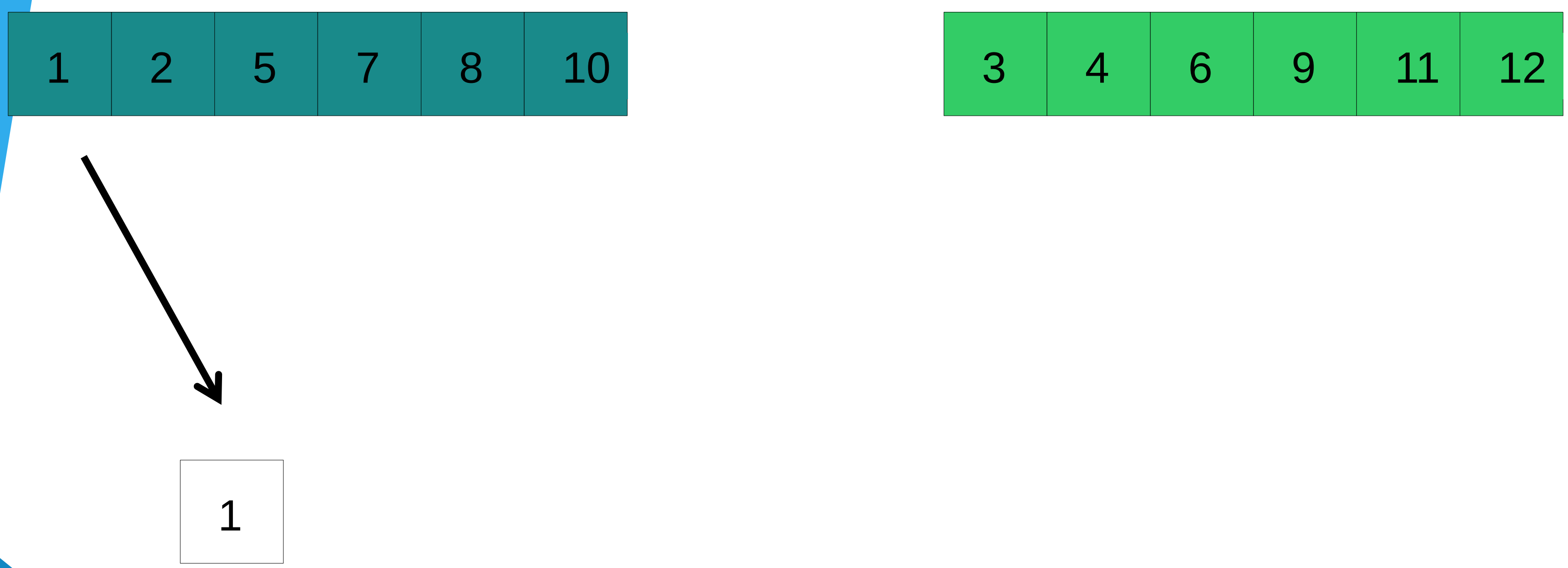
Join both lists with complexity O(n).

1	2	5	7	8	10

3	4	6	9	11	12

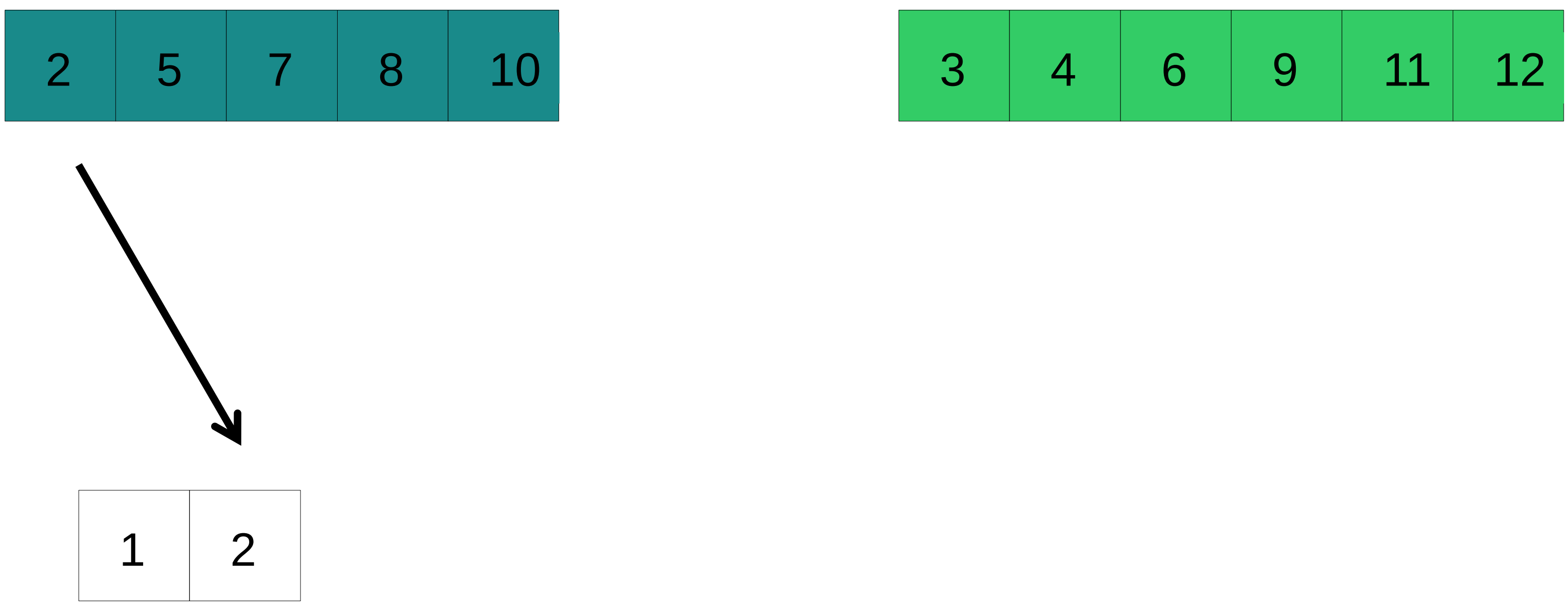
A Parallel Sorting Algorithm

Join both lists with complexity O(n).



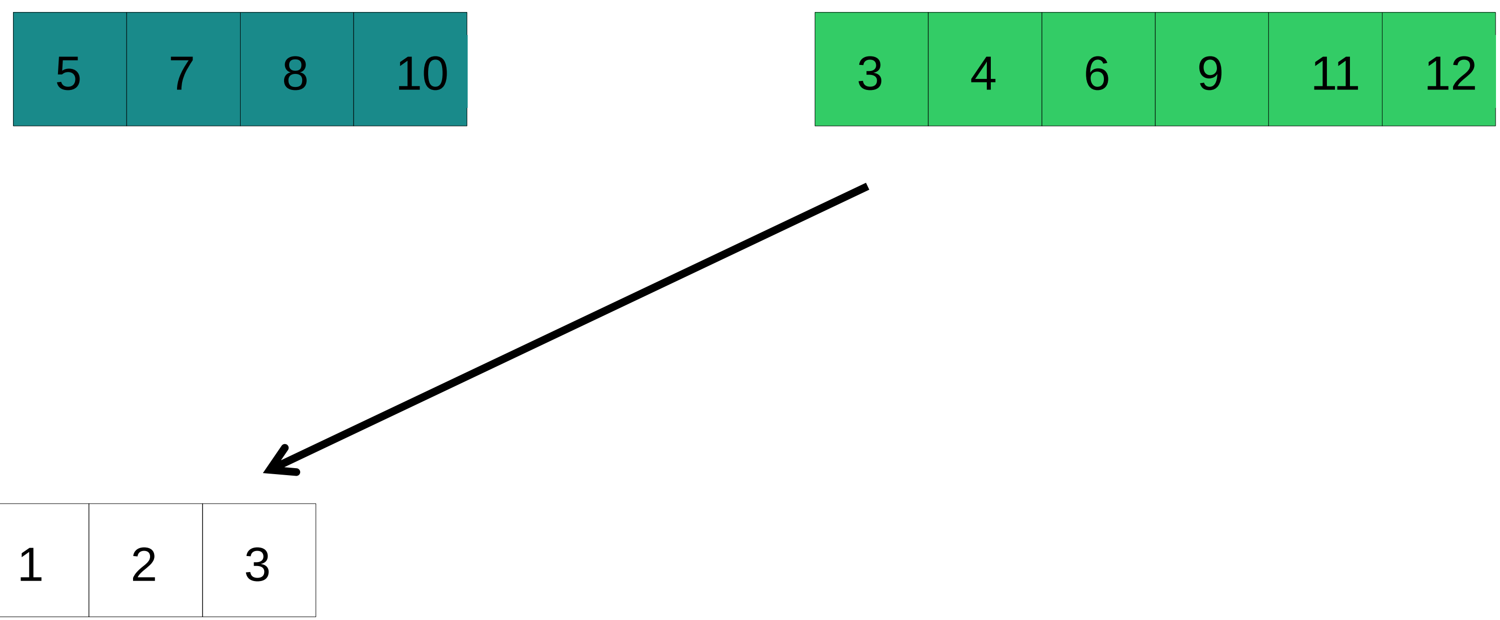
A Parallel Sorting Algorithm

Join both lists with complexity O(n).



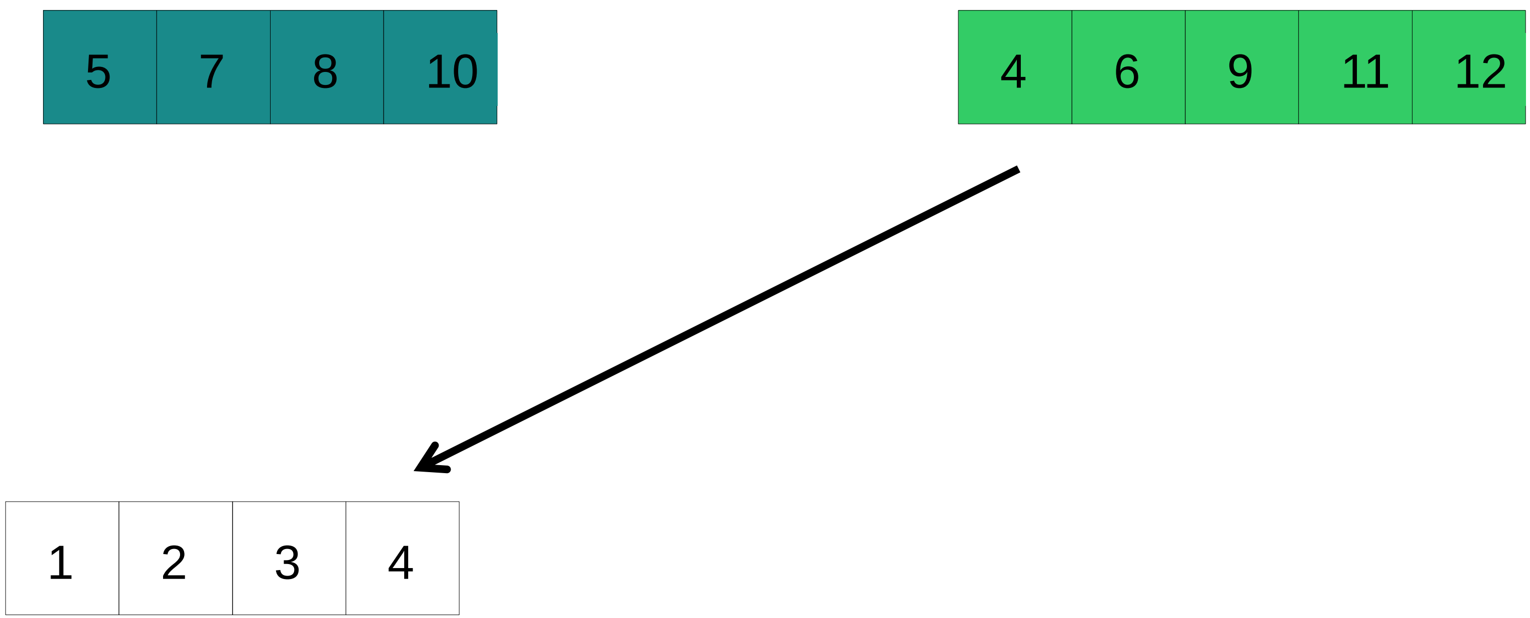
A Parallel Sorting Algorithm

Join both lists with complexity O(n).



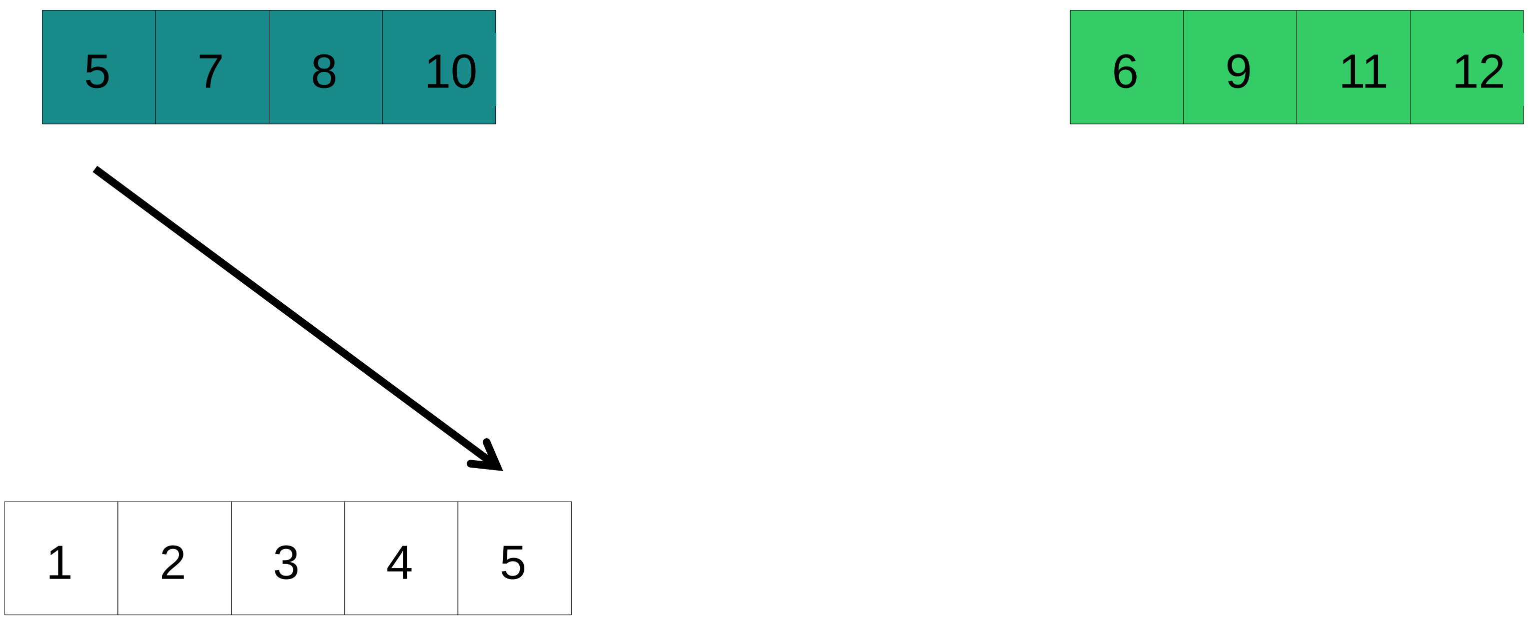
A Parallel Sorting Algorithm

Join both lists with complexity O(n).



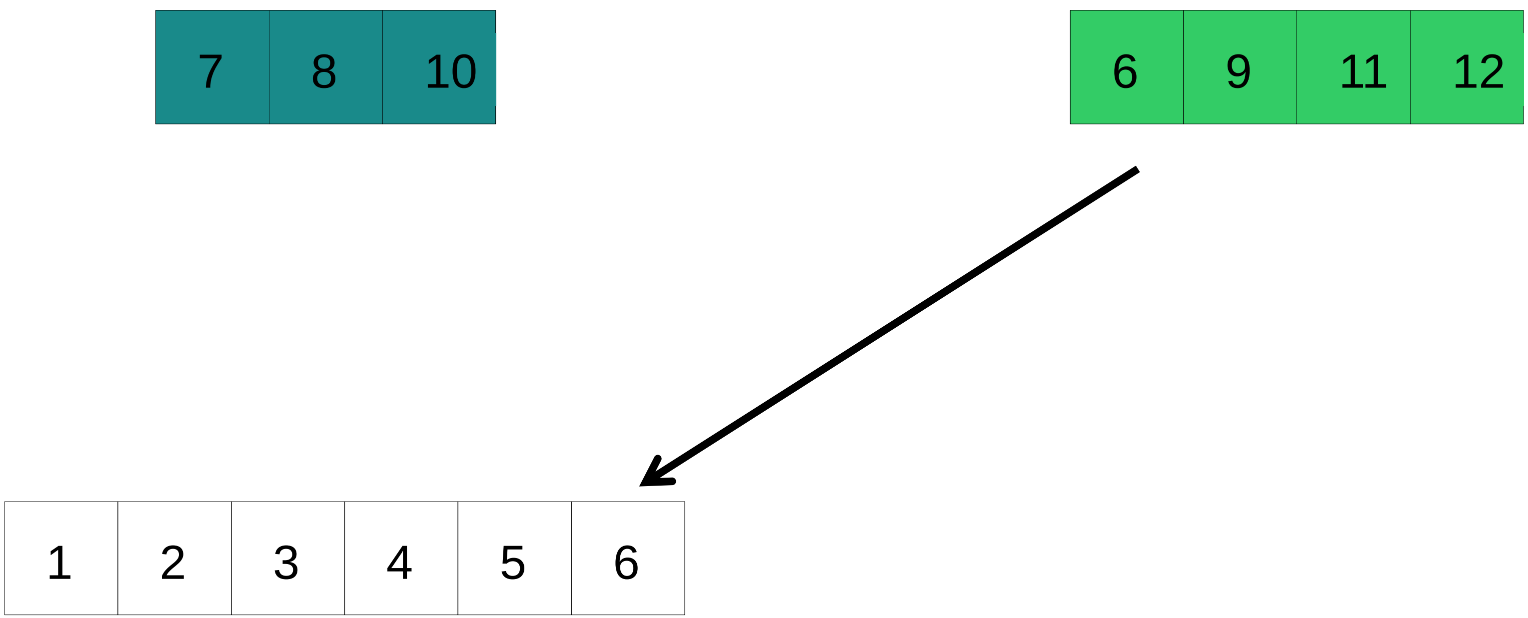
A Parallel Sorting Algorithm

Join both lists with complexity O(n).



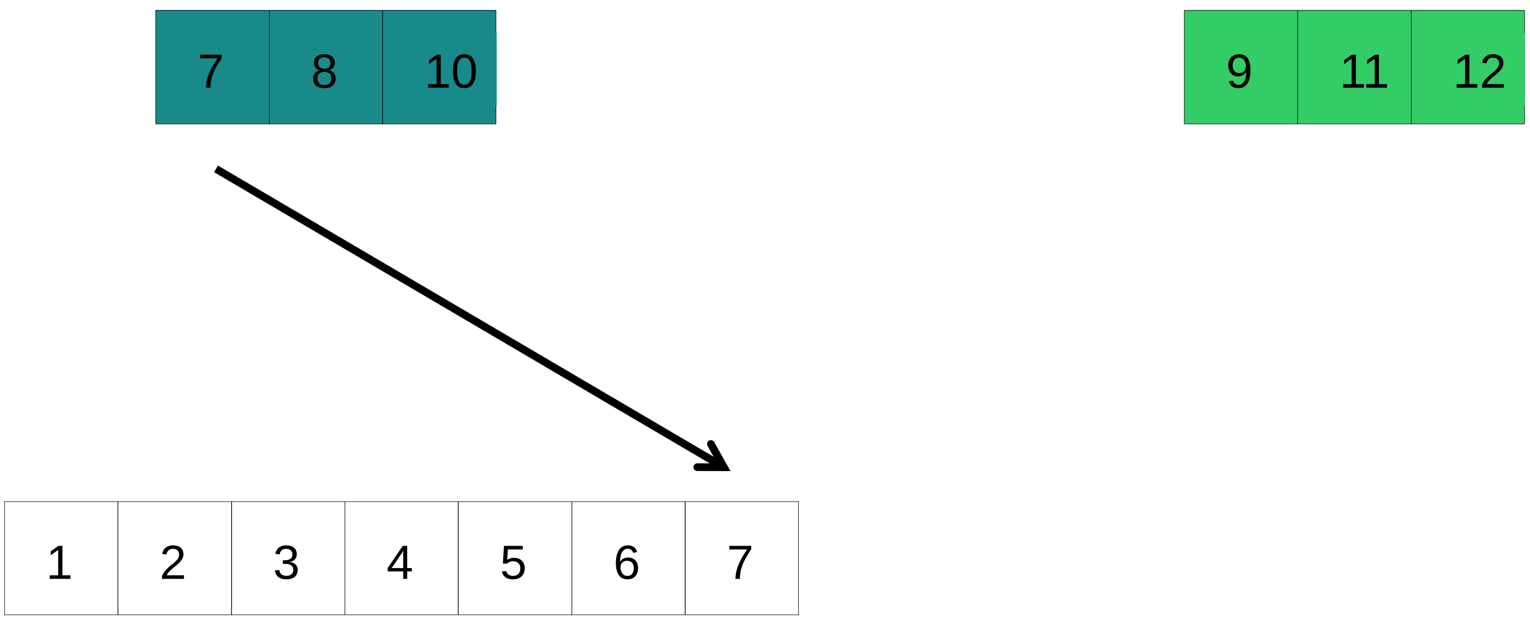
A Parallel Sorting Algorithm

Join both lists with complexity O(n).



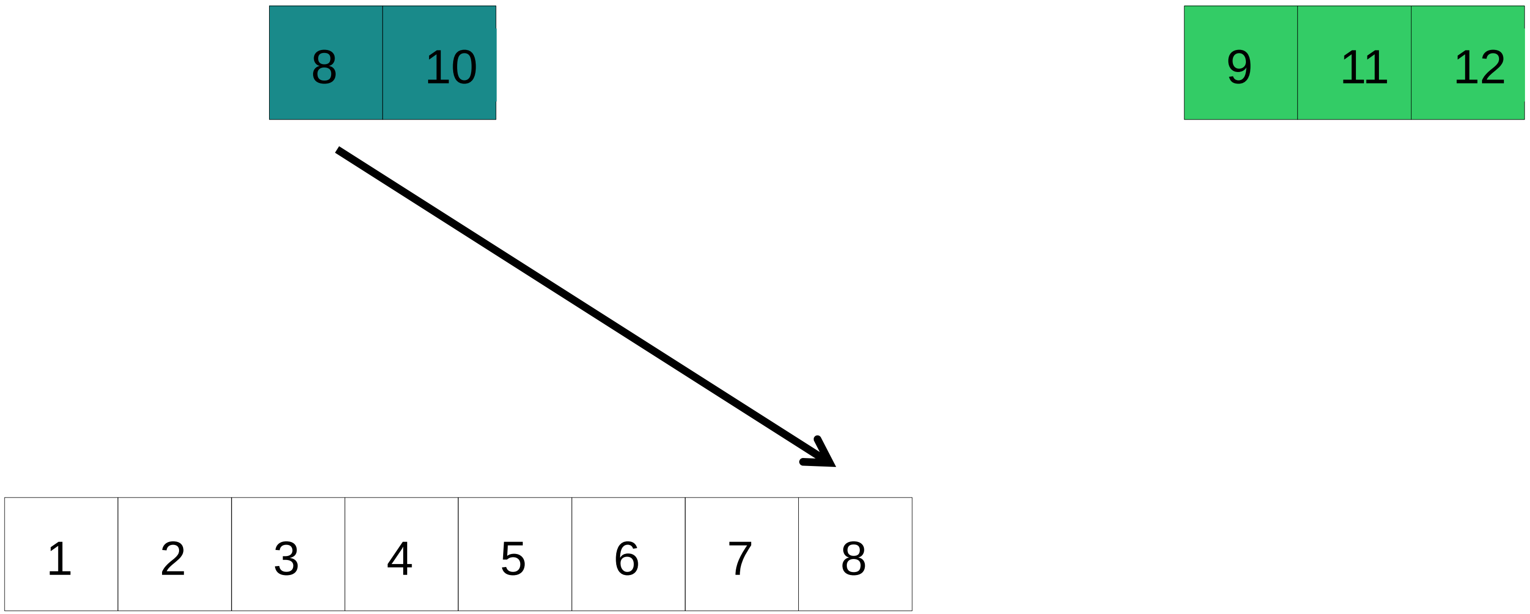
A Parallel Sorting Algorithm

Join both lists with complexity O(n).



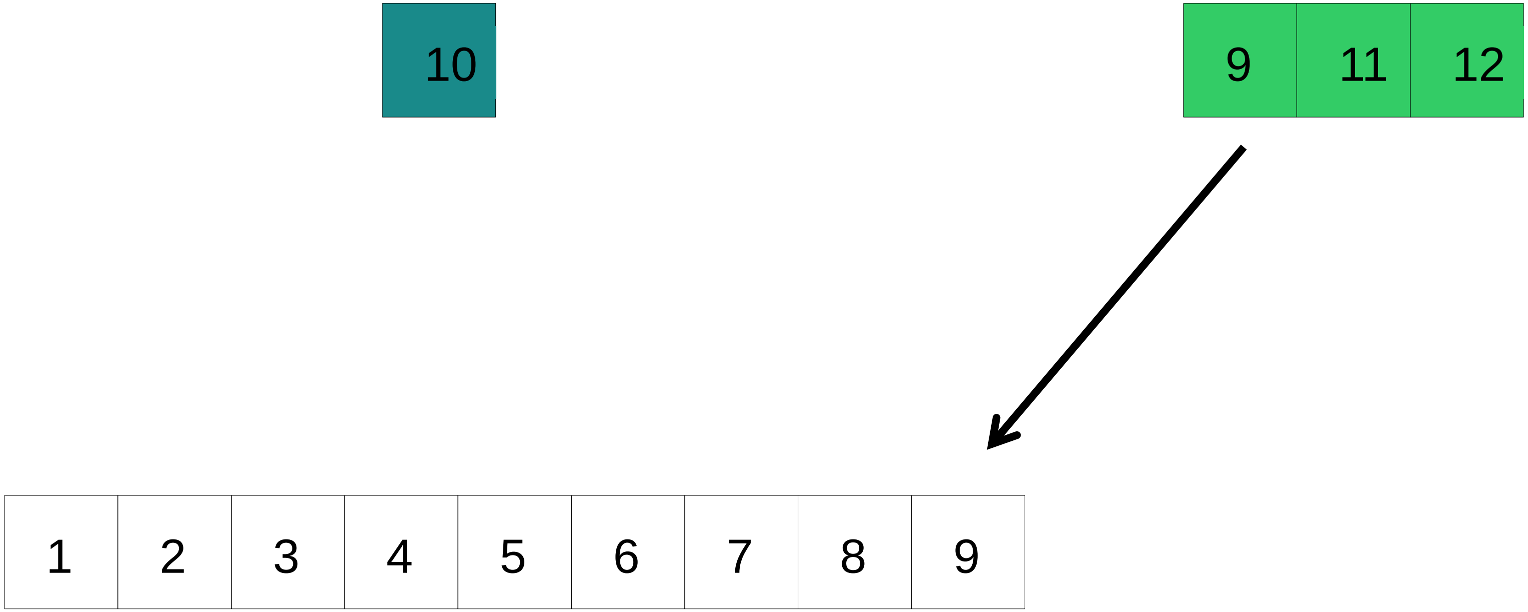
A Parallel Sorting Algorithm

Join both lists with complexity O(n).



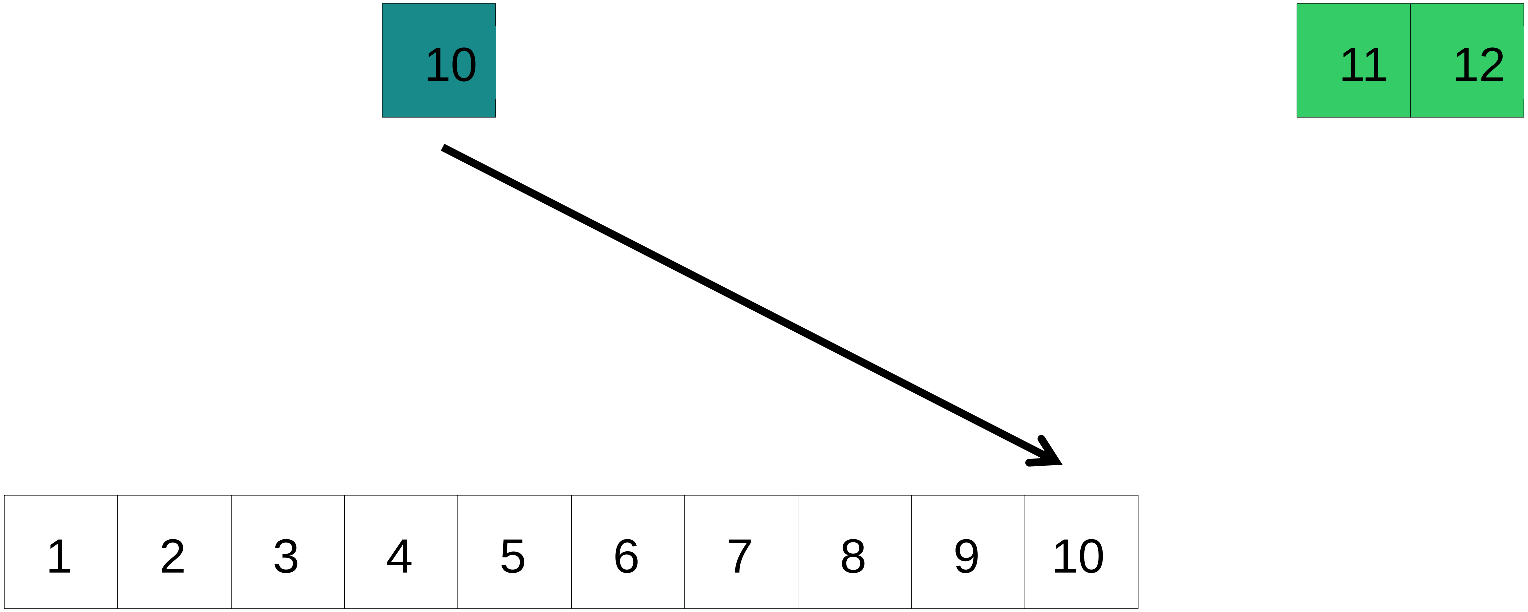
A Parallel Sorting Algorithm

Join both lists with complexity O(n).



A Parallel Sorting Algorithm

Join both lists with complexity O(n).

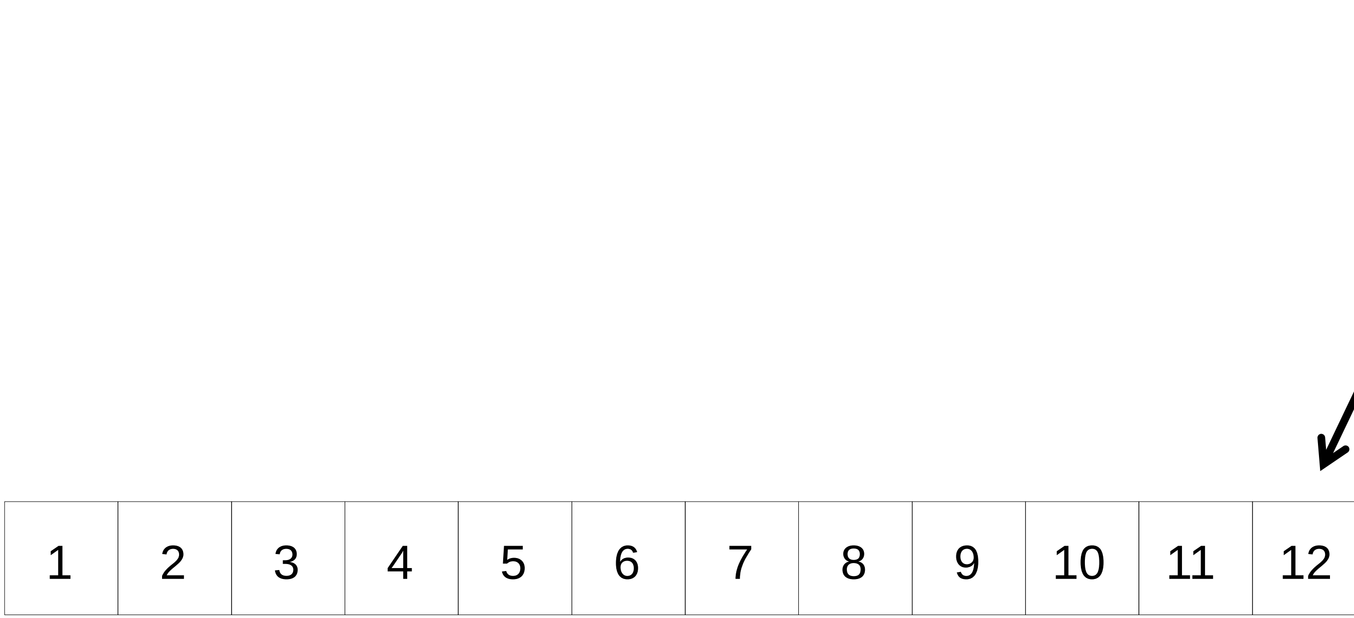


A Parallel Sorting Algorithm

Join both lists with complexity O(n).

A Parallel Sorting Algorithm

Join both lists with complexity O(n).



A Parallel Sorting Algorithm

Join both lists with complexity O(n).

1	2	3	4	5	6	7	8	9	10	11	12

A Parallel Sorting Algorithm

Sort lists

```
…
#pragma omp parallel
{
        int start, local_size;
        local_size = global_size / omp_get_num_thread();
        start = omp_get_thread_num() * local_size;
        final = start + local_size;
        SelectionSort(arr+start, local_size);
        #pragma omp barrier
        #pragma omp master
        mergeSorting(arr, local_size, omp_get_num_threads());
}
```

A Parallel Sorting Algorithm

- Before to join all lists, all of them must be sorted. For this reason the directive *omp barrier* is used.

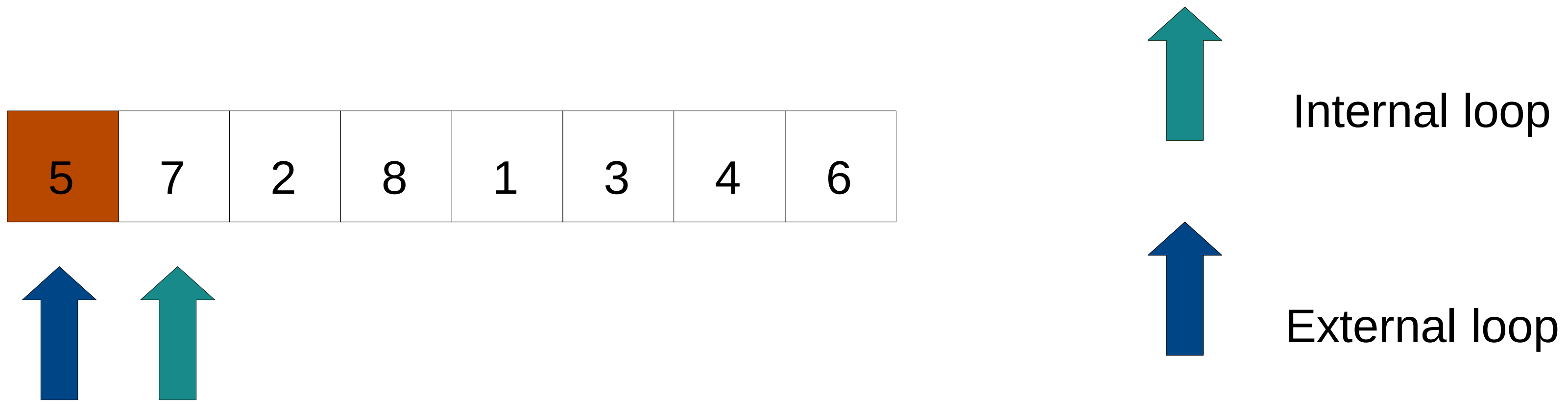
- The algorithm to join the sorted lists is serial, so just one thread is necessary. *omp master* or *omp single* directives can be used.

Selection Sort

```
void SelectionSort (int *array, int size) {
        int i, j, min;
        for (i = 0; i < (size-1); i++) {
                min = i;
                for (j = (i+1); j < size; j++)
                        if(array[j] < array[min])
                                        min = j;
                if (i != min) {
                        int swap = array[i];
                        array[i] = array[min];
                        array[min] = swap;
                }
        }
}
```

External loop: For each iteration, one element is sorted from left to right.
Internal loop: It searches for a smaller element to do an Exchange to the most left position still not sorted.

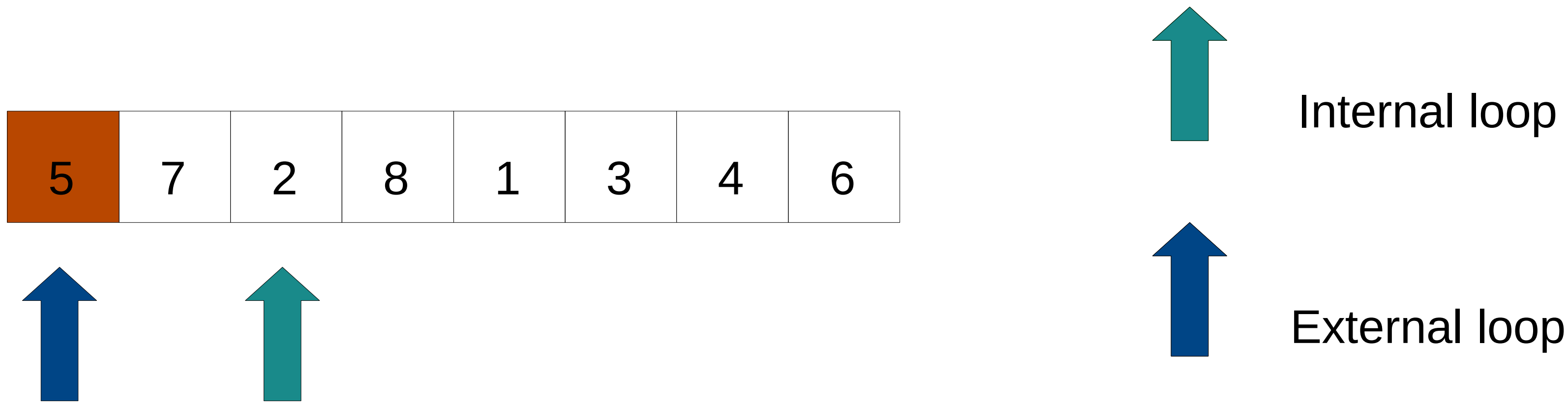
Selection Sort



i	j	min
0	1	0

min: index of smaller number

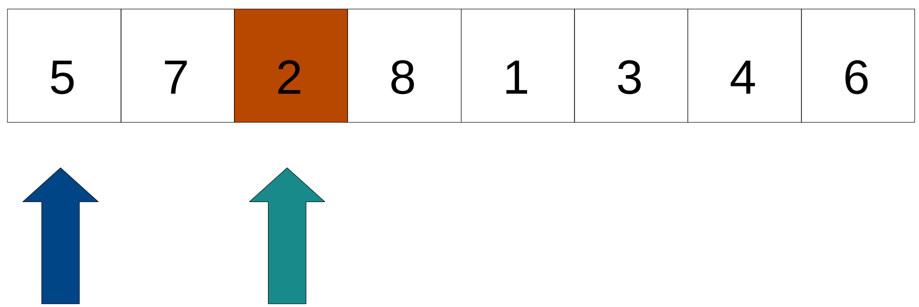
Selection Sort

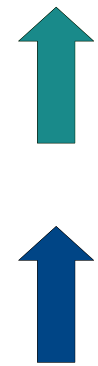


i	j	min
0	2	0

min: index of smaller number

Selection Sort





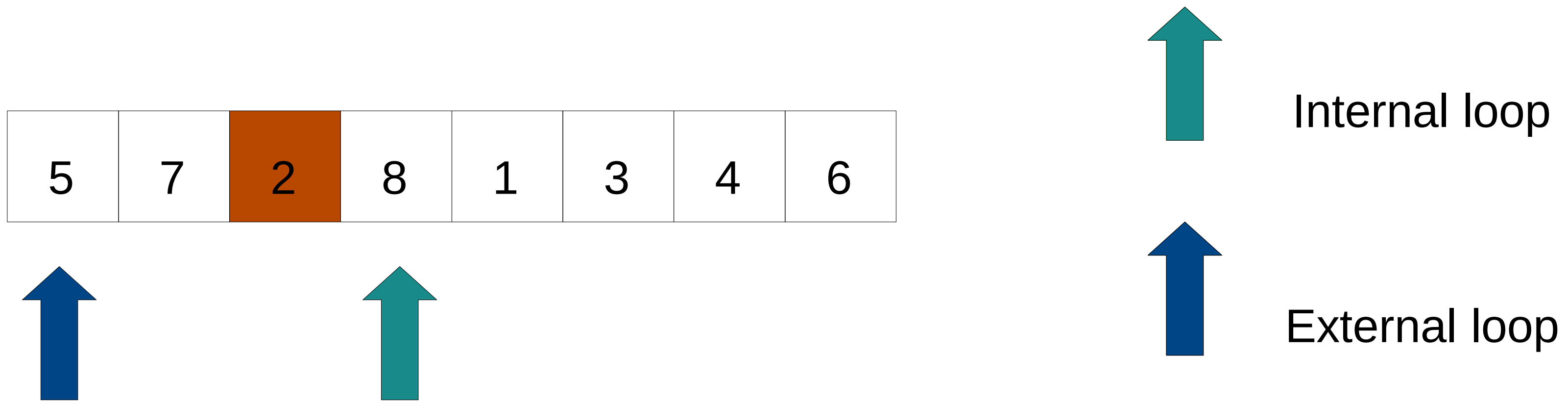
Internal loop

External loop

i	j	min
0	2	2

min: index of smaller number

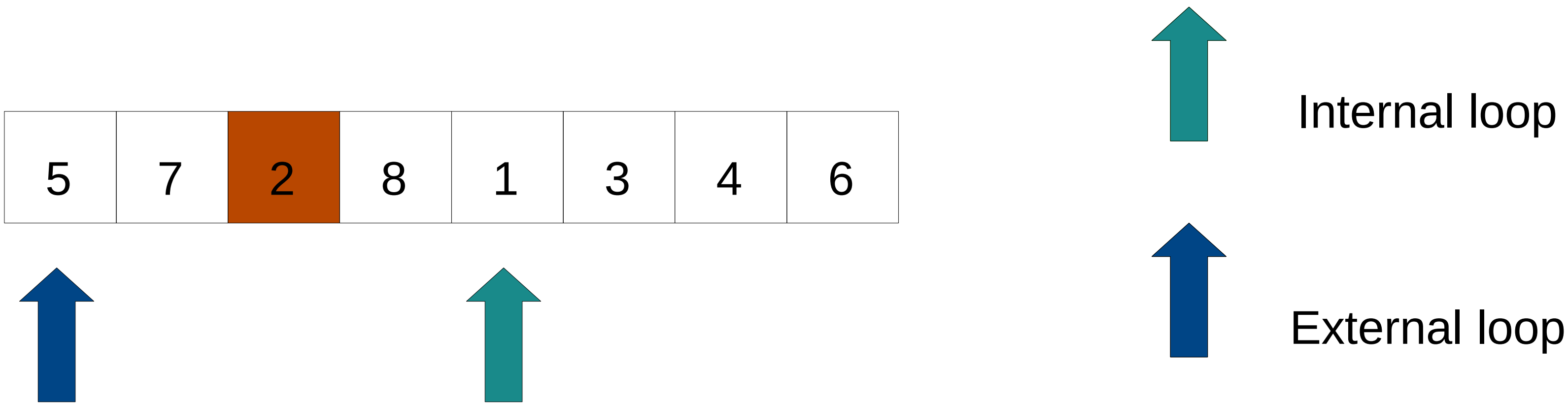
Selection Sort



i	j	min
0	3	2

min: index of smaller number

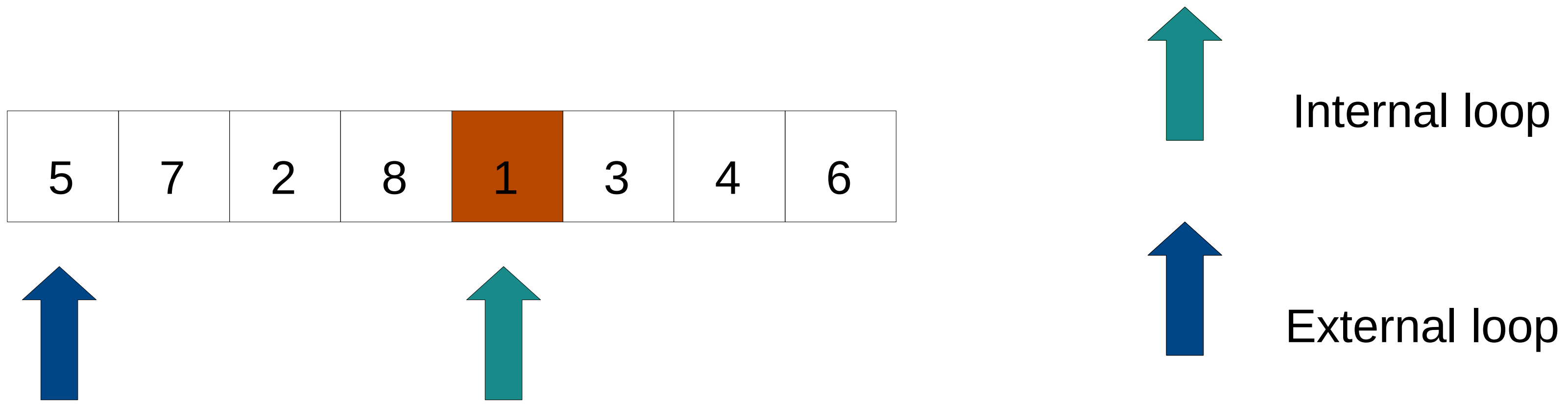
Selection Sort



i	j	min
0	4	2

min: index of smaller number

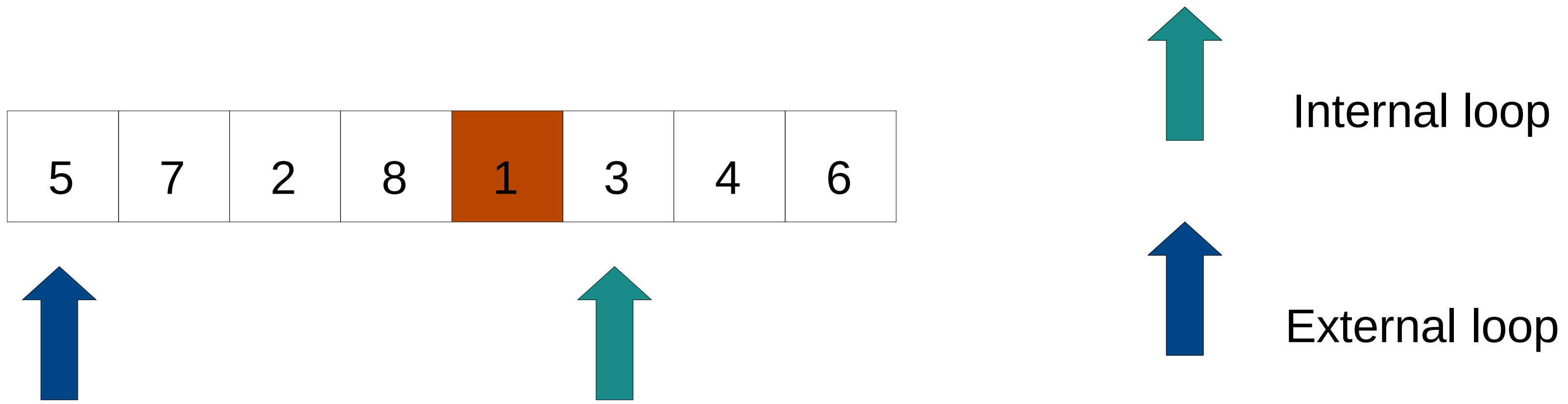
Selection Sort



i	j	min
0	4	4

min: index of smaller number

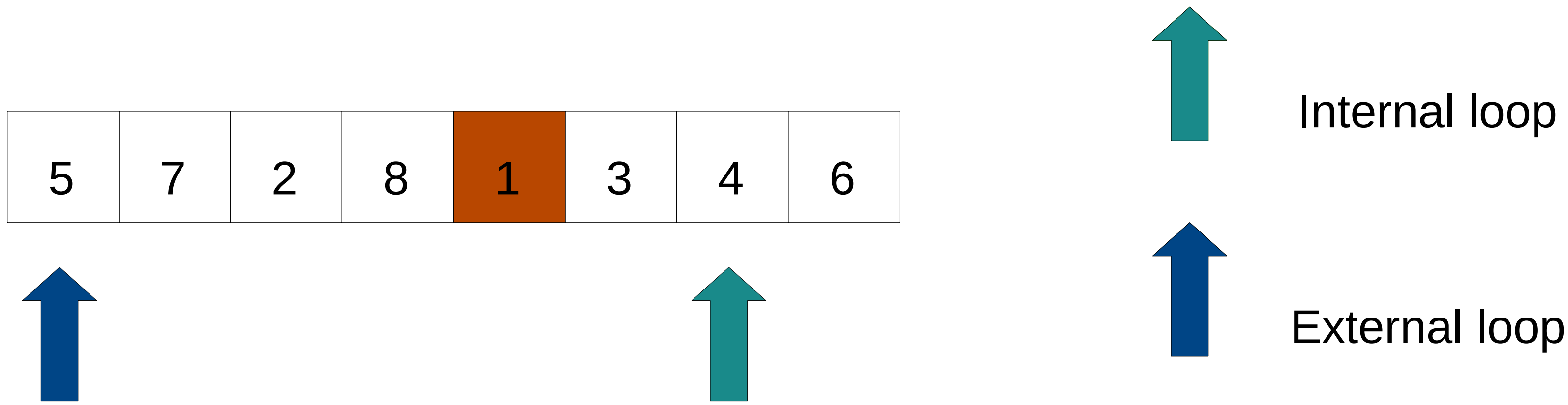
Selection Sort



i	j	min
0	5	4

min: index of smaller number

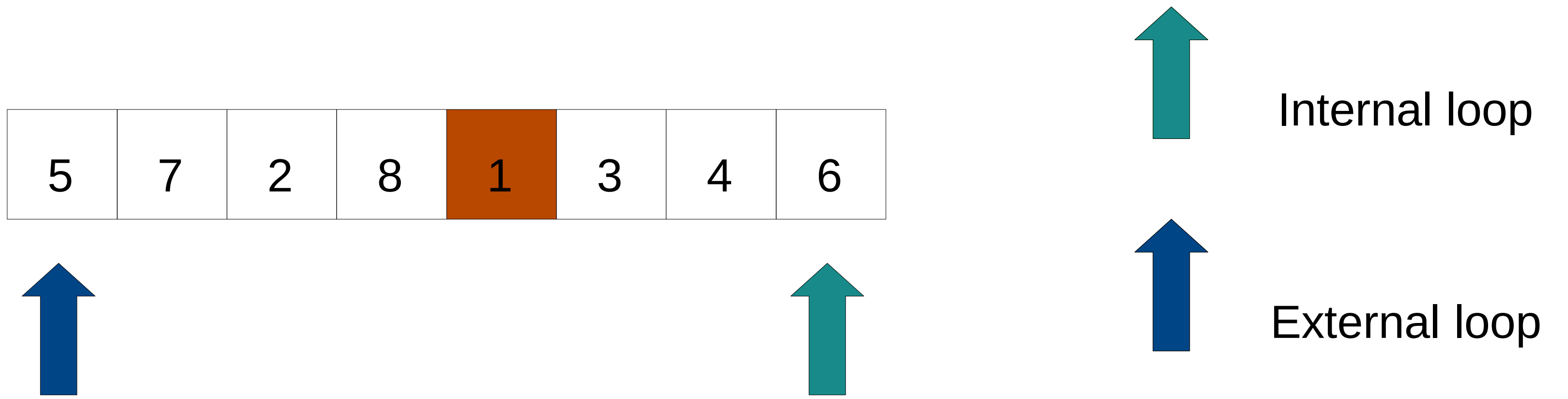
Selection Sort



i	j	min
0	6	4

min: index of smaller number

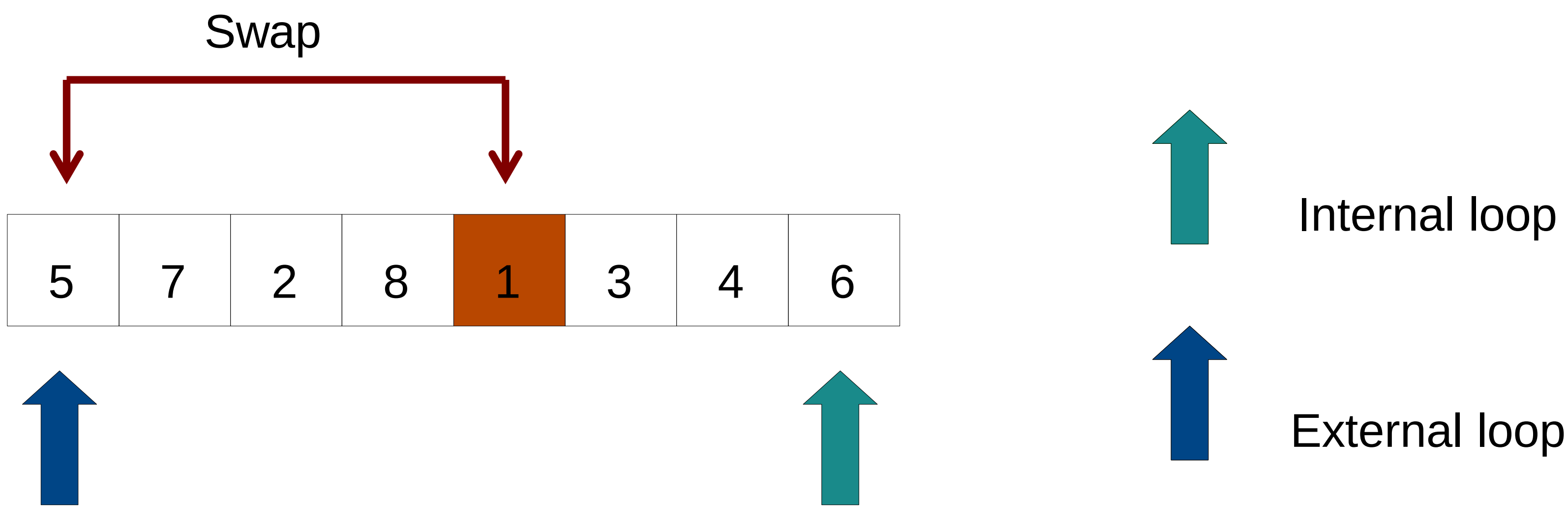
Selection Sort



i	j	min
0	7	4

min: index of smaller number

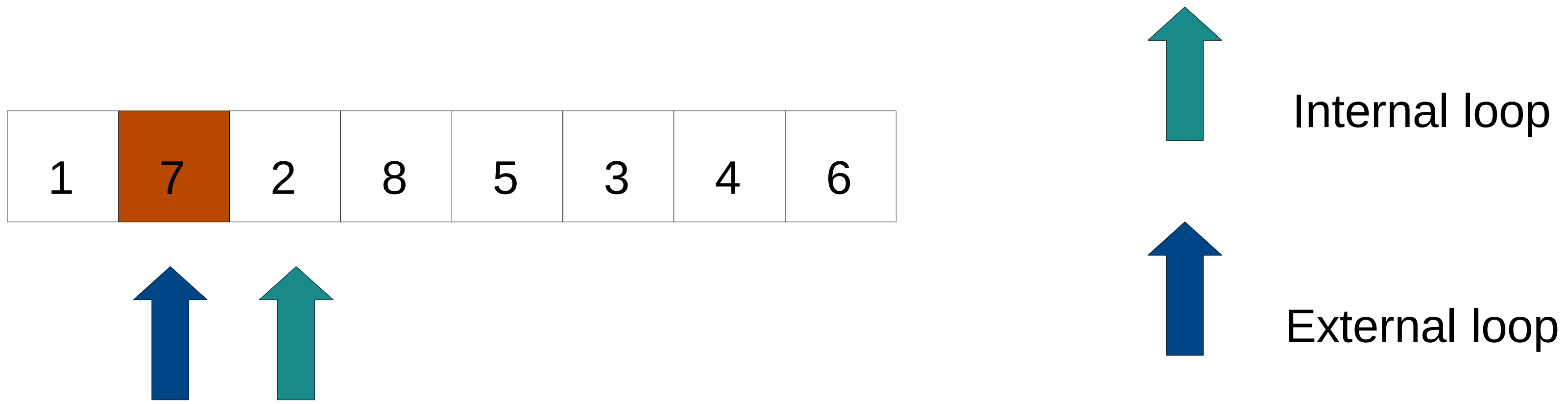
Selection Sort



i	j	min
0	7	4

min: index of smaller number

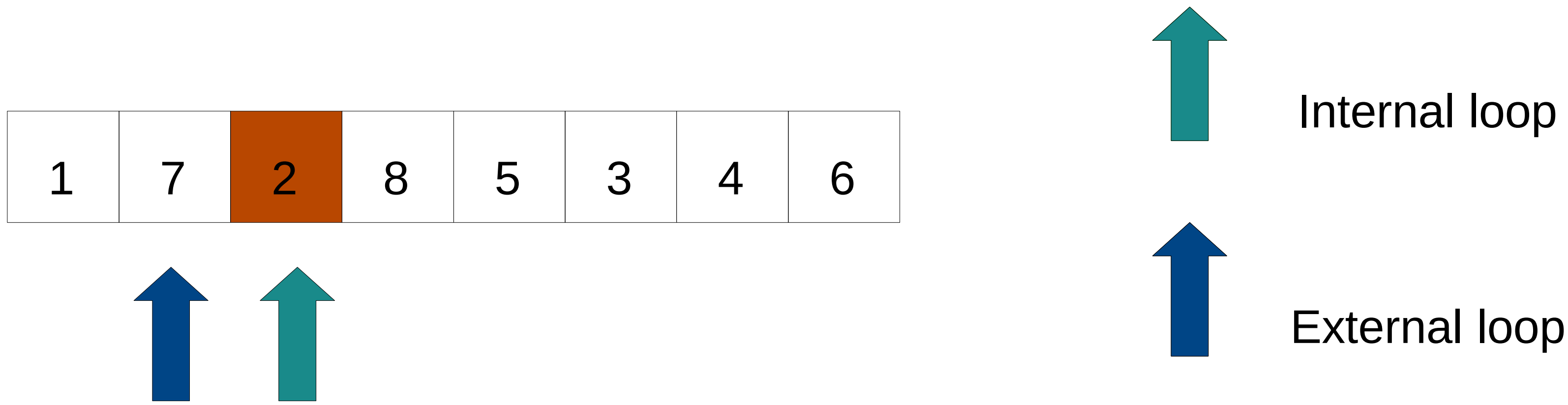
Selection Sort



i	j	min
1	2	1

min: index of smaller number

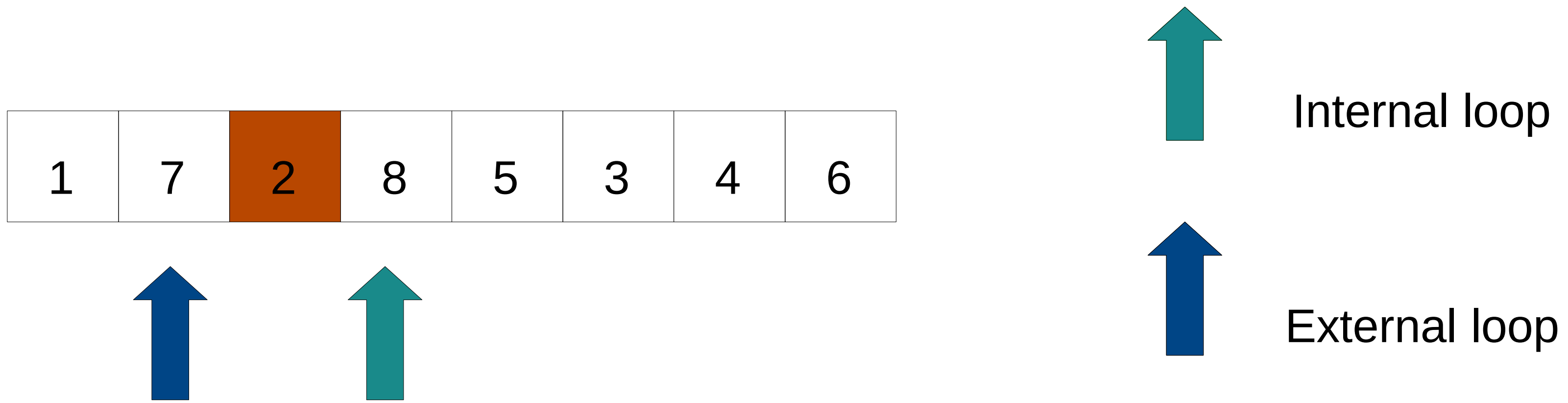
Selection Sort



i	j	min
1	2	2

min: index of smaller number

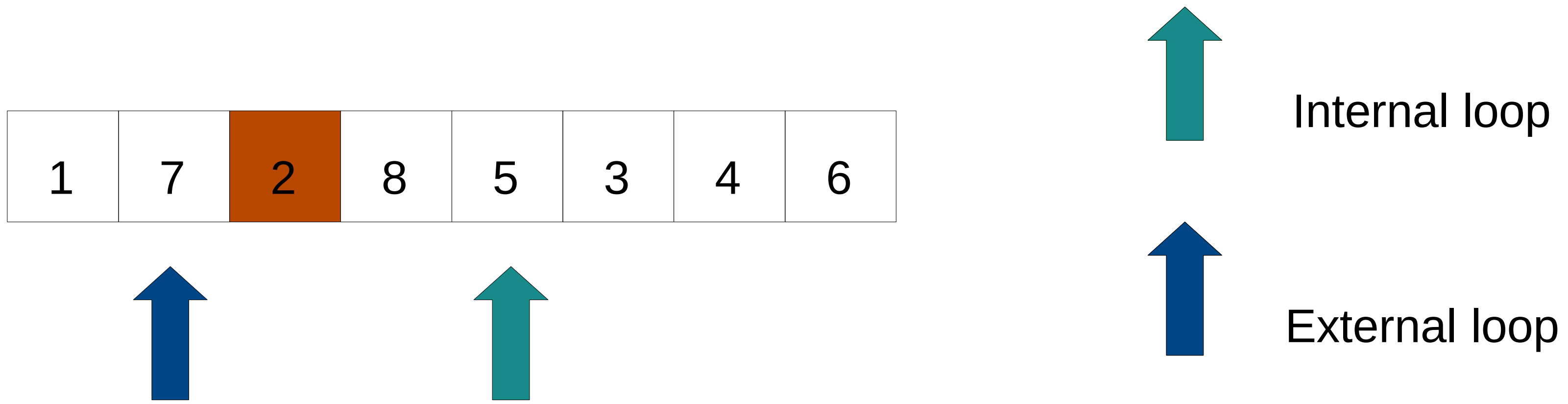
Selection Sort



i	j	min
1	3	2

min: index of smaller number

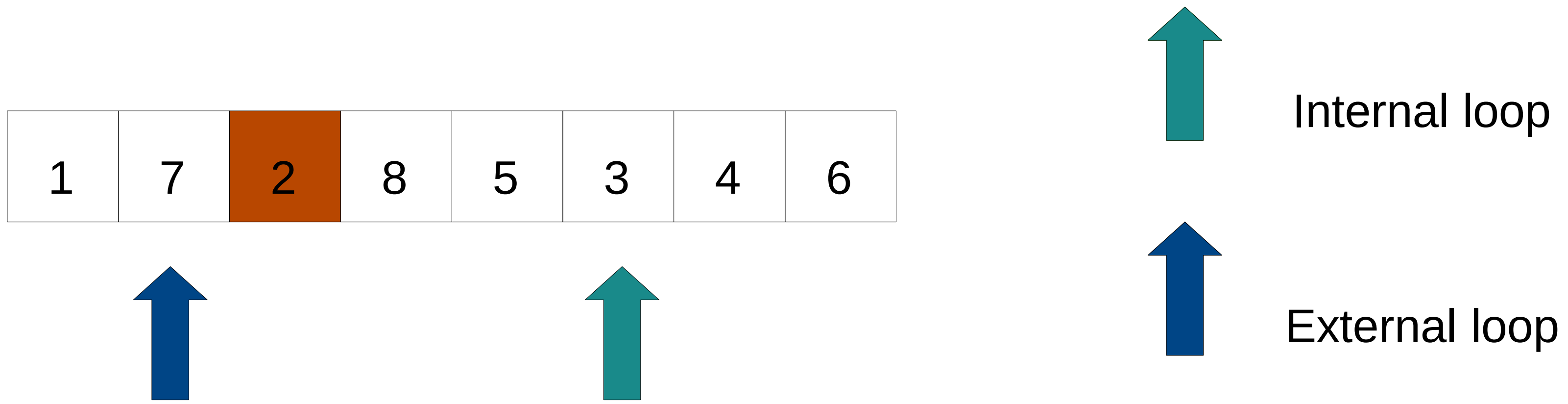
Selection Sort



i	j	min
1	4	2

min: index of smaller number

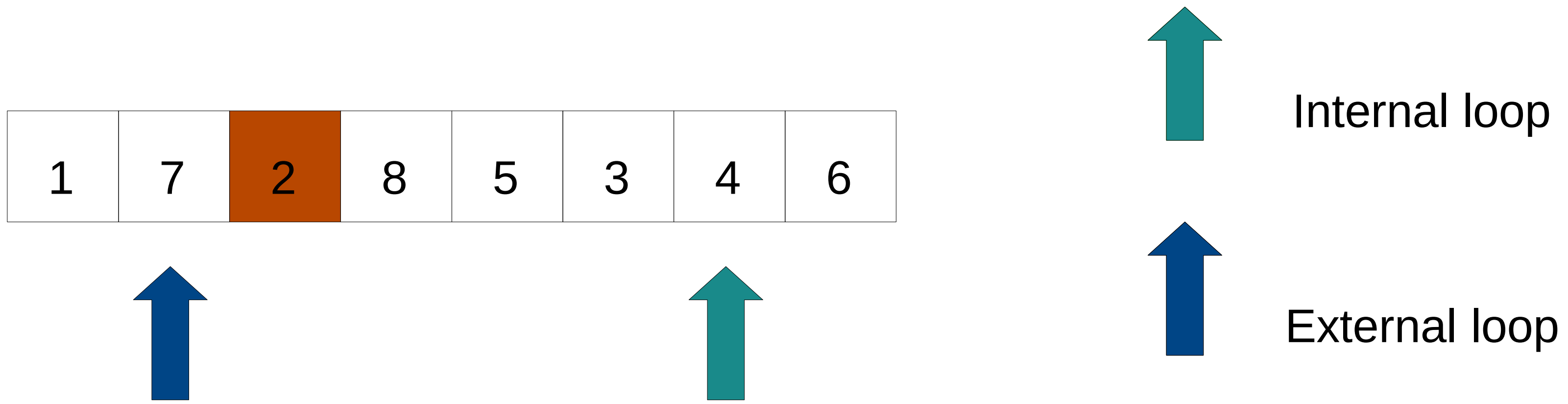
Selection Sort



i	j	min
1	5	2

min: index of smaller number

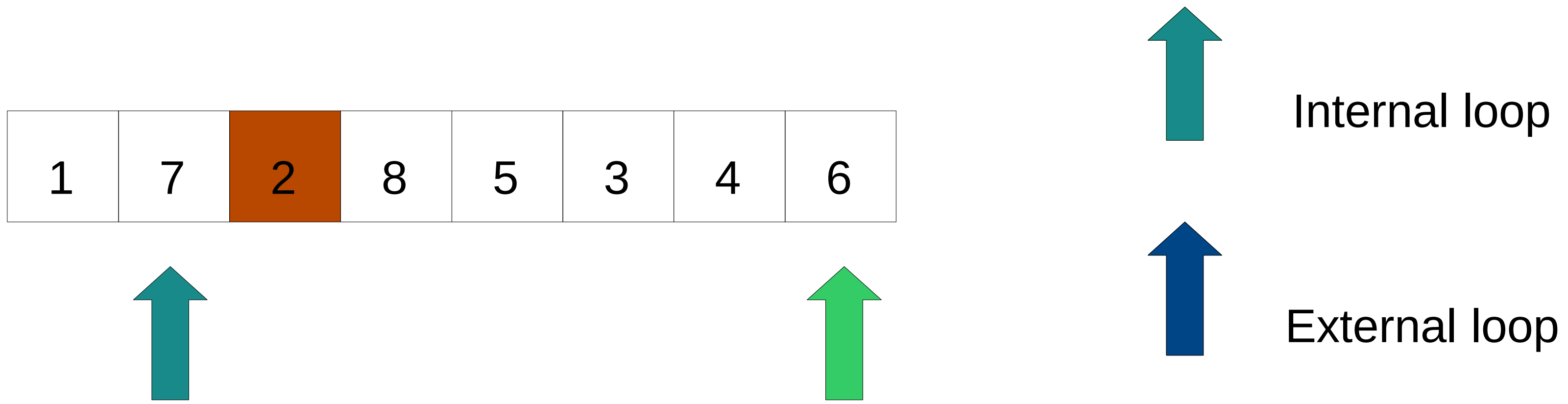
Selection Sort



i	j	min
1	6	2

min: index of smaller number

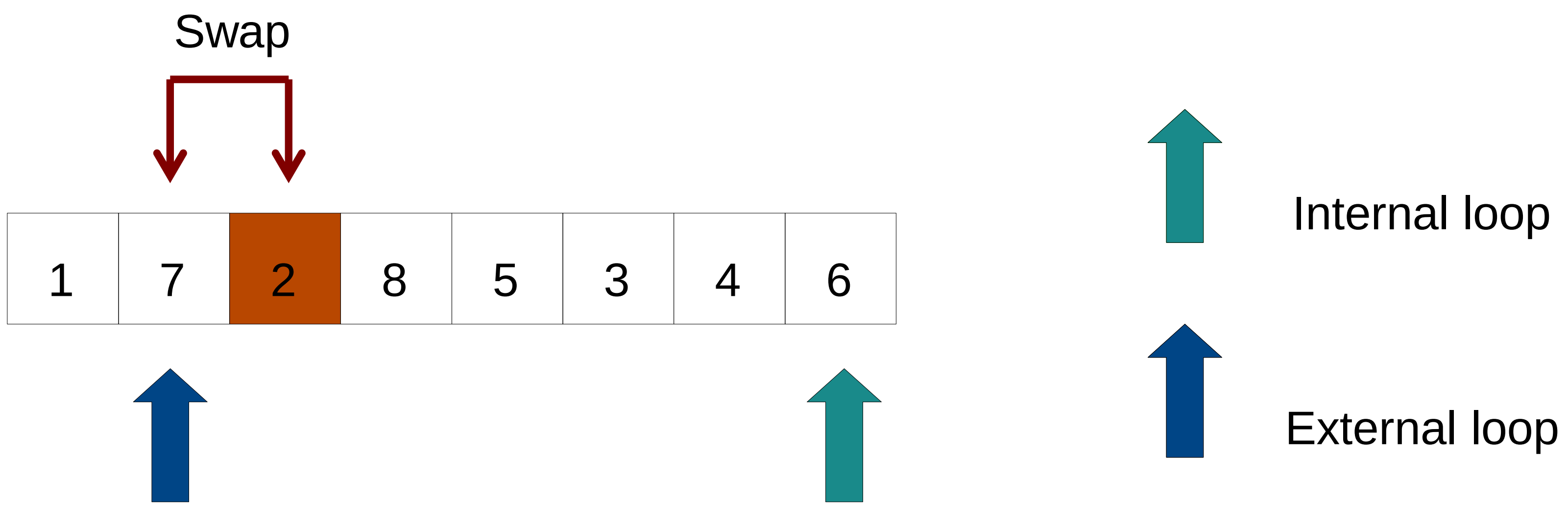
Selection Sort



i	j	min
1	7	2

min: index of smaller number

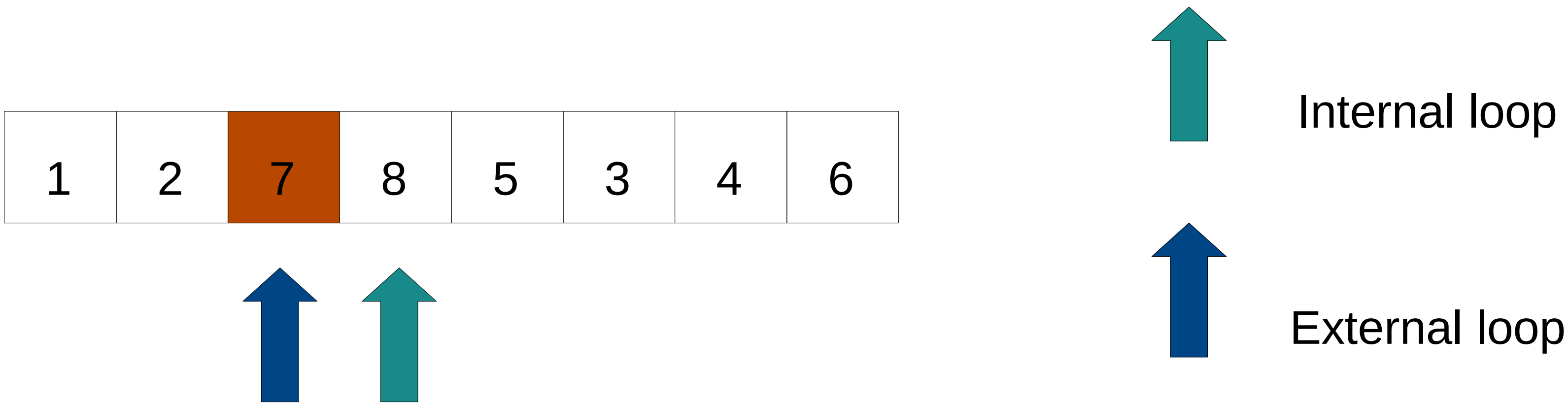
Selection Sort



i	j	min
1	7	2

min: index of smaller number

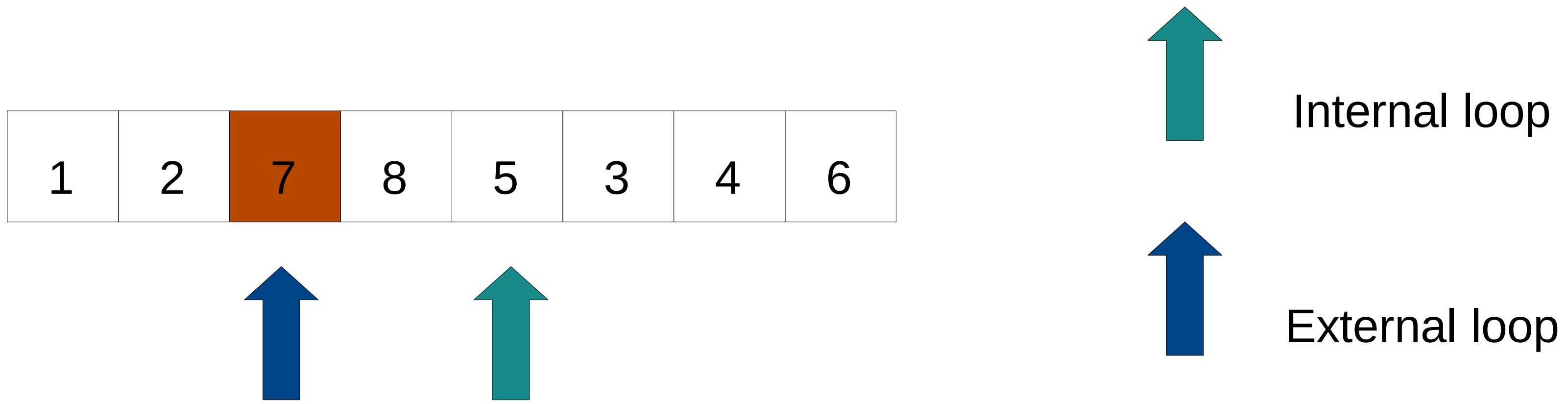
Selection Sort



i	j	min
2	3	2

min: index of smaller number

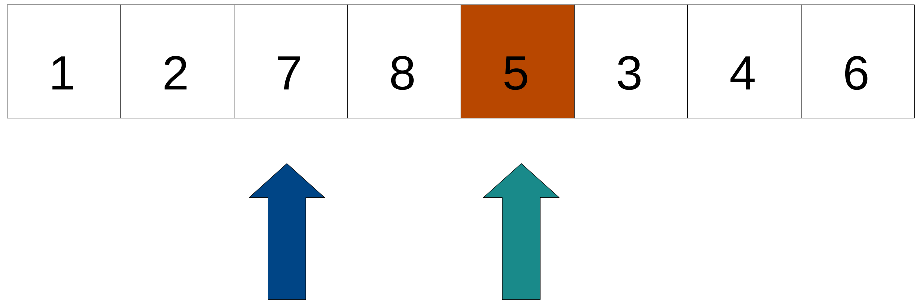
Selection Sort

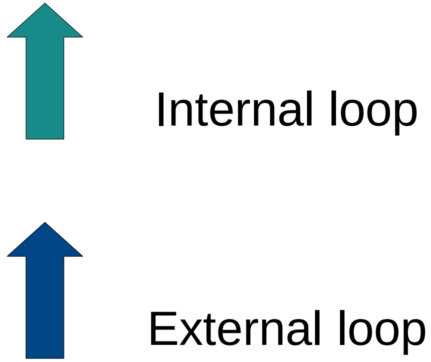


i	j	min
2	4	2

min: index of smaller number

Selection Sort

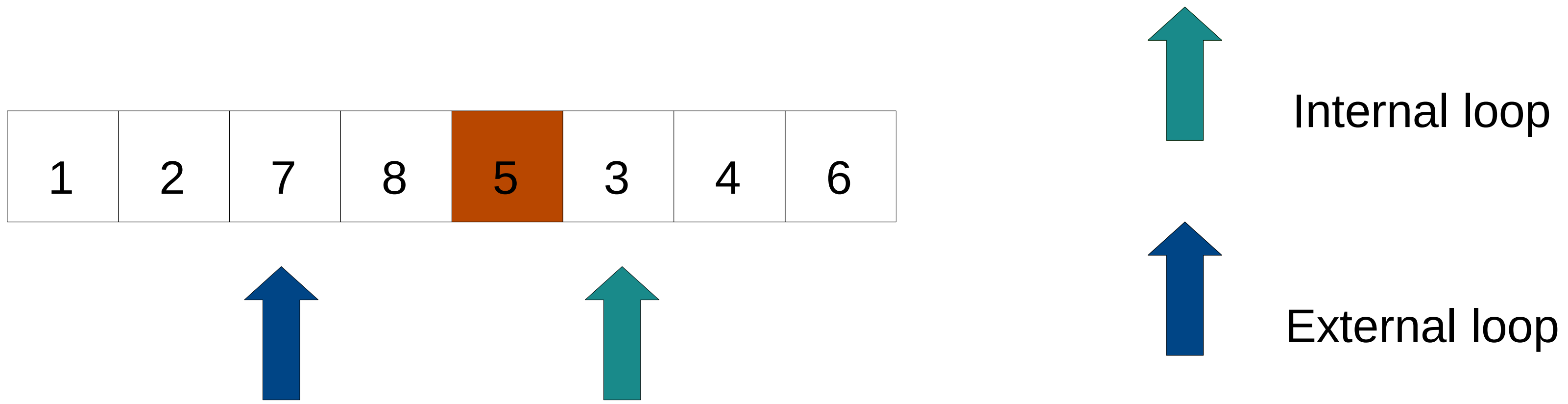




i	j	min
2	4	4

min: index of smaller number

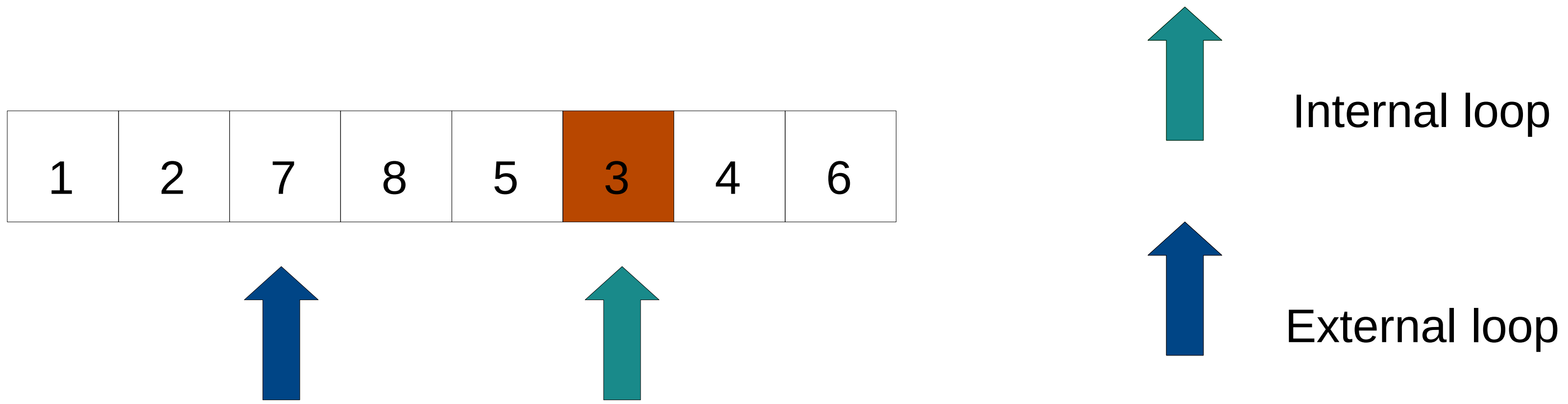
Selection Sort



i	j	min
2	5	4

min: index of smaller number

Selection Sort



i	j	min
2	5	5

min: index of smaller number

Selection Sort

1	2	7	8	5	3	4	6

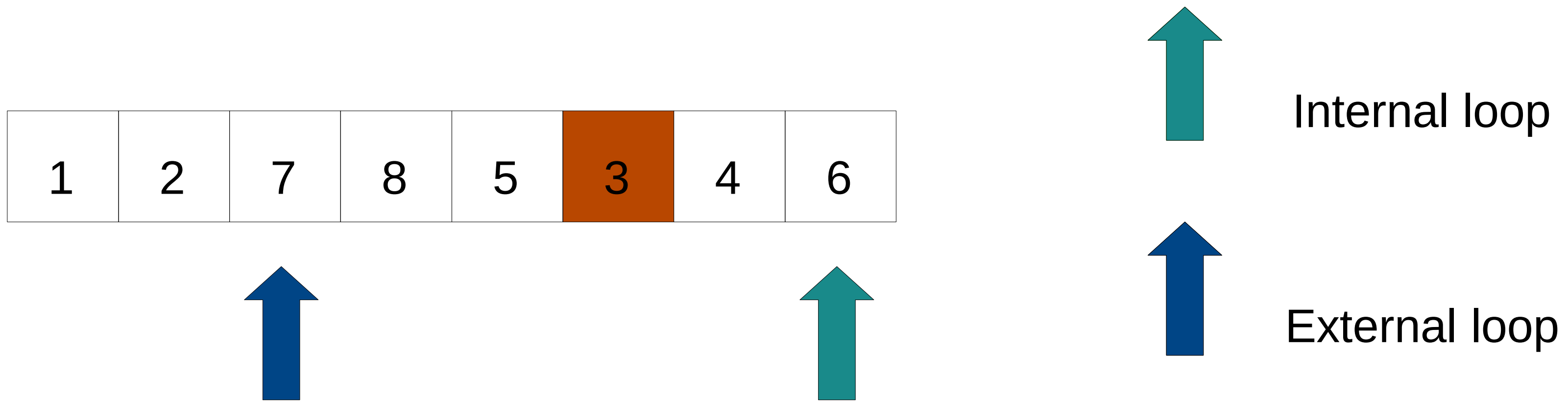
Internal loop

External loop

i	j	min
2	6	5

min: index of smaller number

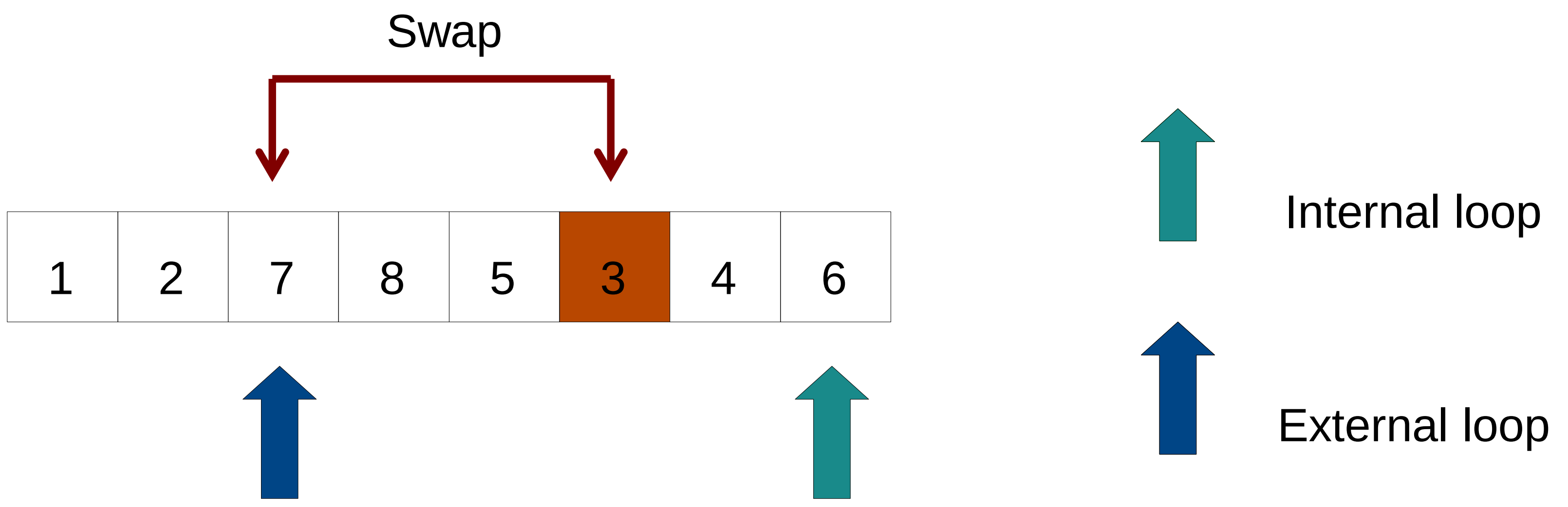
Selection Sort



i	j	min
2	7	5

min: index of smaller number

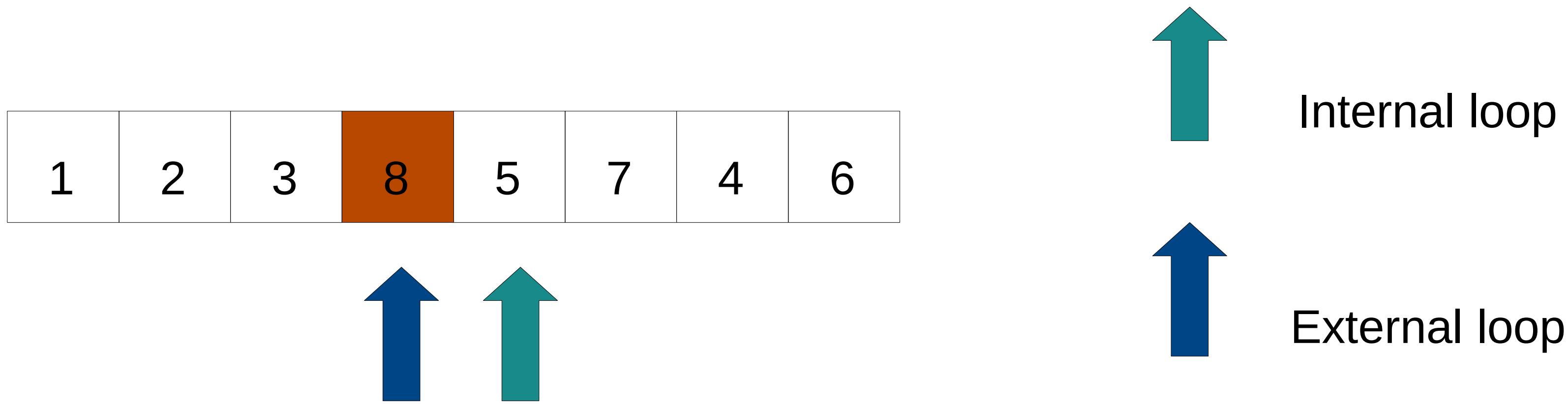
Selection Sort



i	j	min
2	7	5

min: index of smaller number

Selection Sort



i	j	min
3	4	3

min: index of smaller number

Selection Sort

First step:
- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

Selection Sort

First step:
- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

External loop: The next iteration starts after the element exchange from previous iteration. Data dependency!!!

5	7	2	8	1	3	4	6	i = 0
1	7	2	8	5	3	4	6	i = 1
1	2	7	8	5	3	4	6	i = 2
1	2	3	8	5	7	4	6	i = 3

Note: Each iteration must be executed one at a time, since the next iteration is executed after the sorted elements most left.

Selection Sort

First step:

- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

External loop: The next iteration starts after the element exchange from previous iteration. Data dependency!!!

Internal loop: It searches the smaller element still not sorted. More than one thread can be used.

5	7	2	8	1	3	4	6

Thread 0 (j)	Thread 1 (j)	min
1	5	0

Selection Sort

First step:
- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

External loop: The next iteration starts after the element exchange from previous iteration. Data dependency!!!

Internal loop: It searches the smaller element still not sorted. More than one thread can be used.

5	7	2	8	1	3	4	6

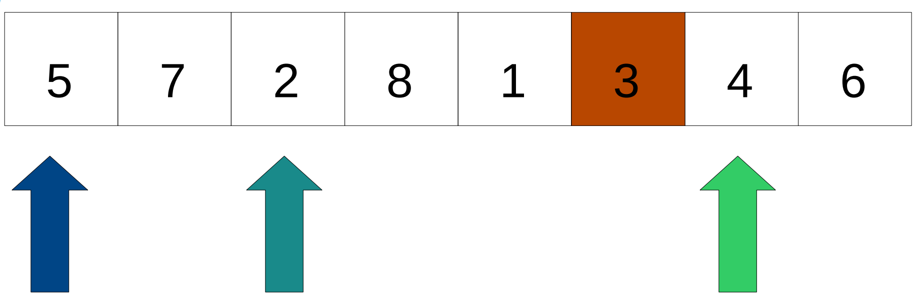
Thread 0 (j)	Thread 1 (j)	min
1	5	5

Selection Sort

First step:
- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

External loop: The next iteration starts after the element exchange from previous iteration. Data dependency!!!

Internal loop: It searches the smaller element still not sorted. More than one thread can be used.



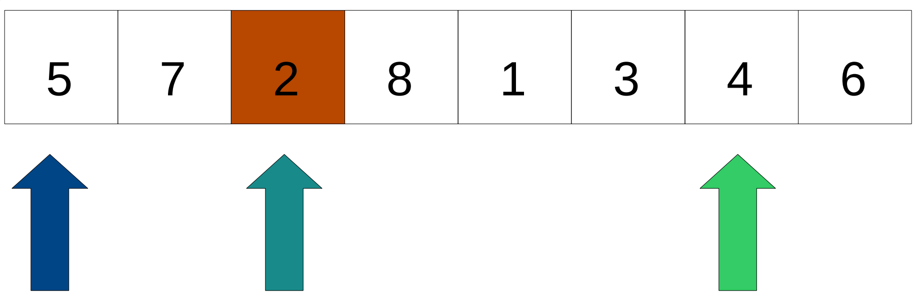
Thread 0 (j)	Thread 1 (j)	min
2	6	5

Selection Sort

First step:
- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

External loop: The next iteration starts after the element exchange from previous iteration. Data dependency!!!

Internal loop: It searches the smaller element still not sorted. More than one thread can be used.



Thread 0 (j)	Thread 1 (j)	min
2	6	2

Selection Sort

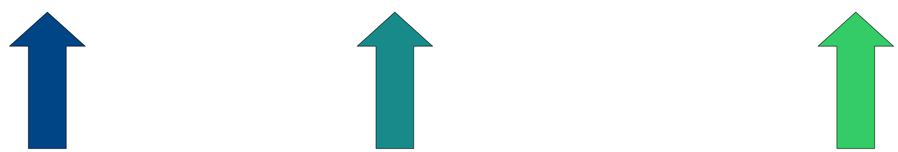
First step:

- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

External loop: The next iteration starts after the element exchange from previous iteration. Data dependency!!!

Internal loop: It searches the smaller element still not sorted. More than one thread can be used.

5	7	2	8	1	3	4	6



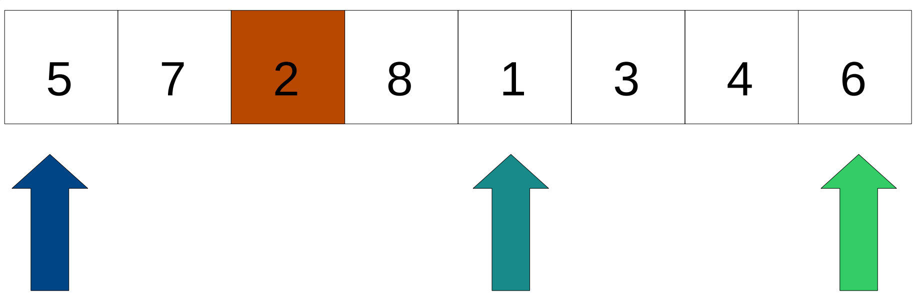
Thread 0 (j)	Thread 1 (j)	min
3	7	2

Selection Sort

First step:
- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

External loop: The next iteration starts after the element exchange from previous iteration. Data dependency!!!

Internal loop: It searches the smaller element still not sorted. More than one thread can be used.



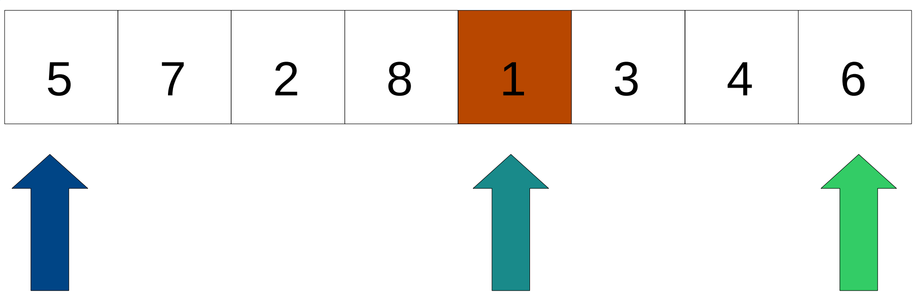
Thread 0 (j)	Thread 1 (j)	min
4	7	2

Selection Sort

First step:
- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

External loop: The next iteration starts after the element exchange from previous iteration. Data dependency!!!

Internal loop: It searches the smaller element still not sorted. More than one thread can be used.



Thread 0 (j)	Thread 1 (j)	min
4	7	4

Selection Sort

```
void SelectionSort (int *array, int size) {
        for (i = 0; i < (n-1); i++) {
                min = i;
                #pragma omp parallel for shared (min) private(j)
                for (j = (i+1); j < n; j++)
                        #pragma omp critical
                        {
                                if(array[j] < array[min])
                                min = j;
                        }
                        if (i != min) {
                                int swap = array[i];
                                array[i] = array[min];
                                array[min] = swap;
                        }
        }
}
```

Selection Sort

```
void SelectionSort (int *array, int size) {
        for (i = 0; i < (n-1); i++) {
                min = i;
                #pragma omp parallel for shared (min) private(j)
                for (j = (i+1); j < n; j++)
                        #pragma omp critical
                        {
                                if(array[j] < array[min])
                                min = j;
                        }
                        if (i != min) {
                                int swap = array[i];
                                array[i] = array[min];
                                array[min] = swap;
                        }
        }
}
```

Note: The algorithm has a complexity O(n²) and the directive *omp critical* inside the internal loop will be also executed O(n²). This directive generates an *overhead*. In practice, this algorithm does not have any scalability when the increase of the number of threads. It can be worse than a serial version.

Selection Sort

Alternatives.

1. Remove the directive *omp critical*?
2. It will generate a race condition, since the variable “min” is shared for all threads.

Selection Sort

Alternatives.
1. Remove the directive *omp critical*?
It will generate a **race condition**, since the variable “min” is shared for all threads.

2. Move *omp critical* to the external loop?
1. Create private variables called “*min_local”.*
2. After the end of the internal loop, “*min”* receives the smallest value among “*min_local” inside a critical region, but outside the internal loop.*
3. *omp critical* will be executed n-1 times.

Selection Sort

Alternatives.

1. Remove the directive *omp critical*?

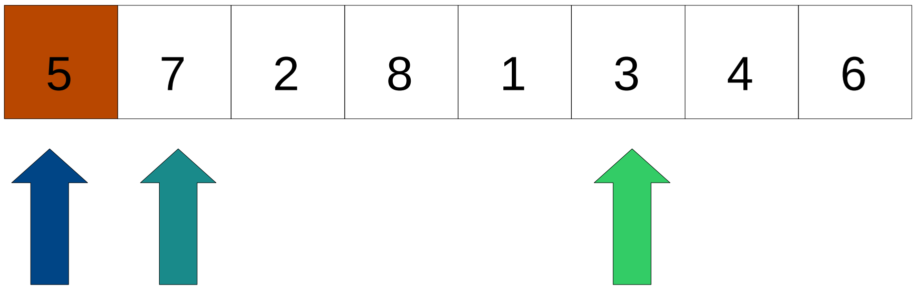
It will generate a **race condition**, since the variable “min” is shared for all threads.

2. Move *omp critical* to the external loop?

1. Create private variables called “*min_local”.*

2. After the end of the internal loop, “*min”* receives the smallest value among “*min_local” inside a critical region, but outside the internal loop.*

3. *omp critical* will be executed n-1 times.



	Thread 0	*Thread* 1
j	1	5
min_local	0	0

Selection Sort

Alternatives.

1. Remove the directive *omp critical*?

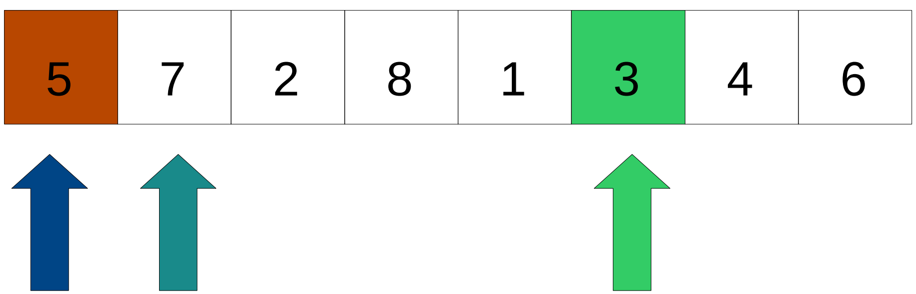
It will generate a **race condition**, since the variable "min" is shared for all threads.

2. Move *omp critical* to the external loop?

1. Create private variables called "*min_local".*

2. After the end of the internal loop, "*min"* receives the smallest value among "*min_local" inside a critical region, but outside the internal loop.*

3. omp critical will be executed n-1 times.



	Thread 0	*Thread* 1
j	1	5
min_local	0	5

Selection Sort

Alternatives.

1. Remove the directive *omp critical*?

It will generate a **race condition**, since the variable "min" is shared for all threads.

2. Move *omp critical* to the external loop?

1. Create private variables called "*min_local*".
2. After the end of the internal loop, "*min*" receives the smallest value among "*min_local*" *inside a critical region, but outside the internal loop.*
3. *omp critical* will be executed n-1 times.

5	7	2	8	1	3	4	6

	Thread 0	*Thread* 1
j	2	6
min_local	0	5

Selection Sort

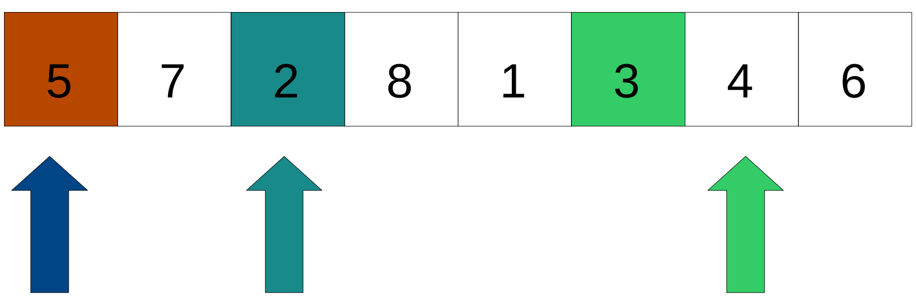
Alternatives.

1. Remove the directive *omp critical*?

It will generate a **race condition**, since the variable “min” is shared for all threads.

2. Move *omp critical* to the external loop?

1. Create private variables called “*min_local”.*
2. After the end of the internal loop, “*min”* receives the smallest value among “*min_local” inside a critical region, but outside the internal loop.*
3. *omp critical* will be executed n-1 times.



	Thread 0	*Thread* 1
j	2	6
min_local	2	5

Selection Sort

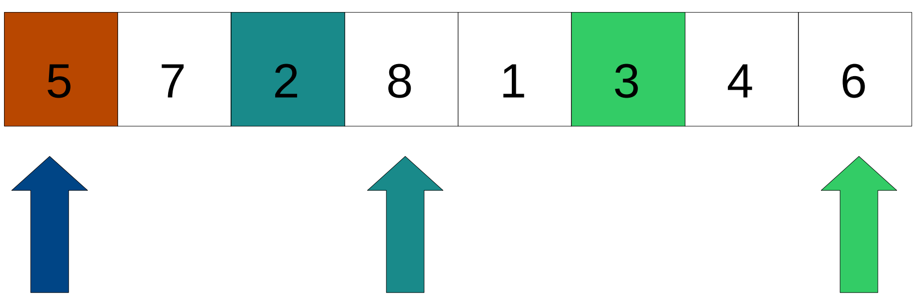
Alternatives.

1. Remove the directive *omp critical*?

It will generate a **race condition**, since the variable “min” is shared for all threads.

2. Move *omp critical* to the external loop?

1. Create private variables called “*min_local”.*
2. After the end of the internal loop, “*min”* receives the smallest value among “*min_local” inside a critical region, but outside the internal loop.*
3. *omp critical* will be executed n-1 times.



	Thread 0	*Thread* 1
j	3	7
min_local	2	5

Selection Sort

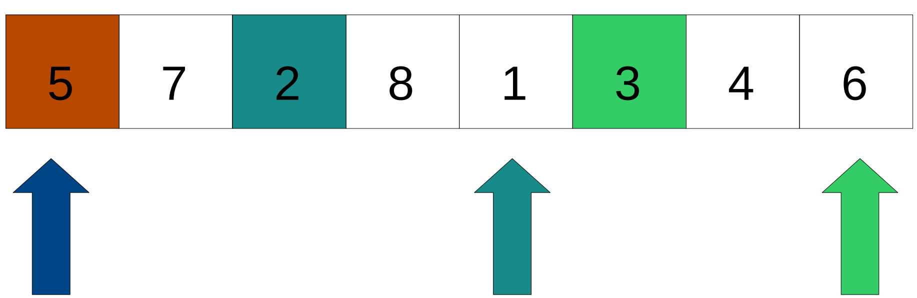
Alternatives.

1. Remove the directive *omp critical*?

It will generate a **race condition**, since the variable “min” is shared for all threads.

2. Move *omp critical* to the external loop?

1. Create private variables called “*min_local”.*
2. After the end of the internal loop, “*min”* receives the smallest value among “*min_local” inside a critical region, but outside the internal loop.*
3. *omp critical* will be executed n-1 times.



	Thread 0	*Thread* 1
j	4	7
min_local	2	5

Selection Sort

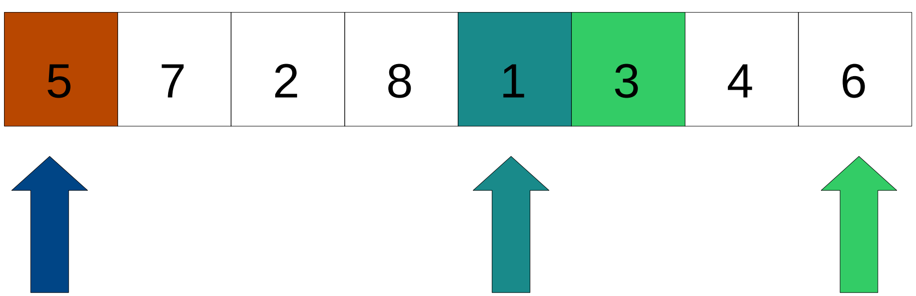
Alternatives.

1. Remove the directive *omp critical*?

It will generate a **race condition**, since the variable “min” is shared for all threads.

2. Move *omp critical* to the external loop?

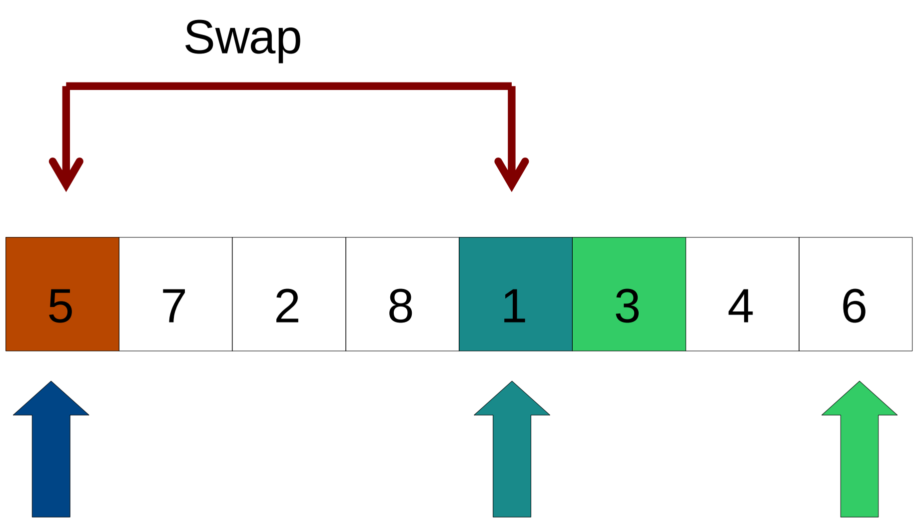
1. Create private variables called “*min_local”.*
2. After the end of the internal loop, “*min”* receives the smallest value among “*min_local” inside a critical region, but outside the internal loop.*
3. *omp critical* will be executed n-1 times.



	Thread 0	*Thread* 1
j	4	7
min_local	4	5

Selection Sort

Note: At the end of internal loop, the two *min_local* will be compared to obtain the smallest value. In the example, thread 1 has an index with the smallest element.



	Thread 0	*Thread* 1
j	4	7
min_local	4	5

Selection Sort

Note: At the end of internal loop, the two *min_local* will be compared to obtain the smallest value. In the example, thread 1 has an index with the smallest element.

1	7	2	8	5	3	4	6

	Thread 0	*Thread* 1
j	2	5
min_local	1	1

Selection Sort

```
void SelectionSortOmp(int *array, int size) {
        int i, j, min;
        for (i = 0; i < (size-1); i++) {
                min = i;
                #pragma omp parallel
                {
                        int local_min = i;
                        #pragma omp for
                        for (j = (i+1); j < size; j++){
                                if(array[j] < array[local_min])
                                        local_min = j;
                        }
                        #pragma omp critical
                        if(array[local_min] < array[min])
                                min = local_min;
                }
                if (i != min) {
                        int swap = array[i];
                        array[i] = array[min];
                        array[min] = swap;
                }
        }
}
```

Parallel Region

Critical region executed n-1 times.

Shell Sort

```
void shellsort(int arr[], int n)
{
    int gap, i, j, temp;
            for (gap = n/2; gap > 0; gap /= 2)
                    for(i = gap; i < n; i++) {
                            int key = arr[i];
                            j = i - gap;
                            while (j >= 0 && arr[j] > key){
                                    arr[j+gap] = arr[j];
                                    j-=gap;
                            }
                            arr[j+gap] = key;
                    }
}
```

- The Shell Sort separates the elements into groups as sublists.
- The number of groups is equal to the value of "*gap*". In the last external loop it will be just one group since the "*gap*" value is 1.
- An adpated Insertion sort is used to sort each group.

Shell Sort

```
void shellsort(int arr[], int n)
{
    int gap, i, j, temp;
        for (gap = n/2; gap > 0; gap /= 2)
            for(i = gap; i < n; i++) {
                int key = arr[i];
                j = i - gap;
                while (j >= 0 && arr[j] > key){
                    arr[j+gap] = arr[j];
                    j-=gap;
                }
                arr[j+gap] = key;
            }

}
```

- The Shell Sort separates the elements into groups as sublists.
- The number of groups is equal to the value of "*gap*". In the last external loop it will be just one group since the "*gap*" value is 1.
- An adpated Insertion sort is used to sort each group.

Shell Sort

Before	5	7	2	8	1	3	4	6
After	1	3	2	6	5	7	4	8

Gap = 8/2 = 4

Shell Sort

Before	5	7	2	8	1	3	4	6
After	1	3	2	6	5	7	4	8

Gap = 8/2 = 4

Before	1	3	2	6	5	7	4	8
After	1	3	2	6	4	7	5	8

Gap = 4/2 = 2

Shell Sort

Gap = 8/2 = 4

Before	5	7	2	8	1	3	4	6
After	1	3	2	6	5	7	4	8

Gap = 4/2 = 2

Before	1	3	2	6	5	7	4	8
After	1	3	2	6	4	7	5	8

Gap = 2/2 = 1

Before	1	3	2	6	5	7	4	8
After	1	2	3	4	5	6	7	8

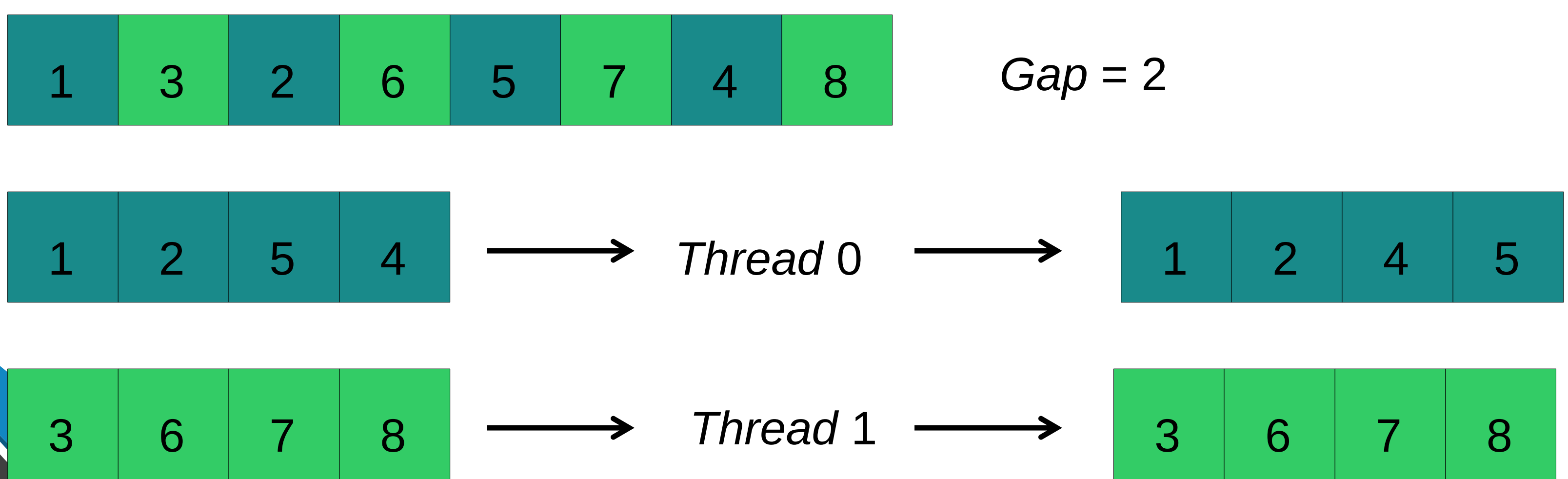
Shell Sort

First step:
- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

Shell Sort

First step:
- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

The Insertion Sort algorithm has a serial behavior, in other words, the instructions must be executed in order. The elements of the same group cannot be exchanged concurrently. However, an element exchange in different groups simultaneously is possible, since there is no dependency between lists.



Shell Sort

External loop descreasing the **gap**

```
void shellsort(int arr[], int n)
{
        int gap, i, j, grupoId, temp;
        for (gap = n/2; gap > 0; gap /= 2){
                ...
        }
}
```

Shell Sort

```
void shellsort(int arr[], int n)
{
        int gap, i, j, grupoId, temp;
        for (gap = n/2; gap > 0; gap /= 2){
                #pragma omp parallel for private(j, i)
                for(grupoId = 0; grupoId < gap; grupoId++) {
                        ...
                }
        }
}
```

A parallel loop can execute **n** threads concurrently. **n** is equal to the number of groups defined by “gap”. If “gap” is equal to 2, then it will be two concurrent threads.

Shell Sort

Each thread will process only the elements of your own.

```
void shellsort(int arr[], int n)
{
        int gap, i, j, grupoId, temp;
        for (gap = n/2; gap > 0; gap /= 2)
                #pragma omp parallel for ptivate(j, i)
                for(grupoId = 0; grupoId < gap; grupoId++)
                        for (i=gap+grupoId; i<n-grupoId; i+=gap) {
                                int key = arr[i];
                                j = i - gap;
                                while (j >= 0 && arr[j] > key) {
                                        arr[j+gap] = arr[j];
                                        j-=gap;
                                }

                                arr[j+gap] = key;
                        }
}
```

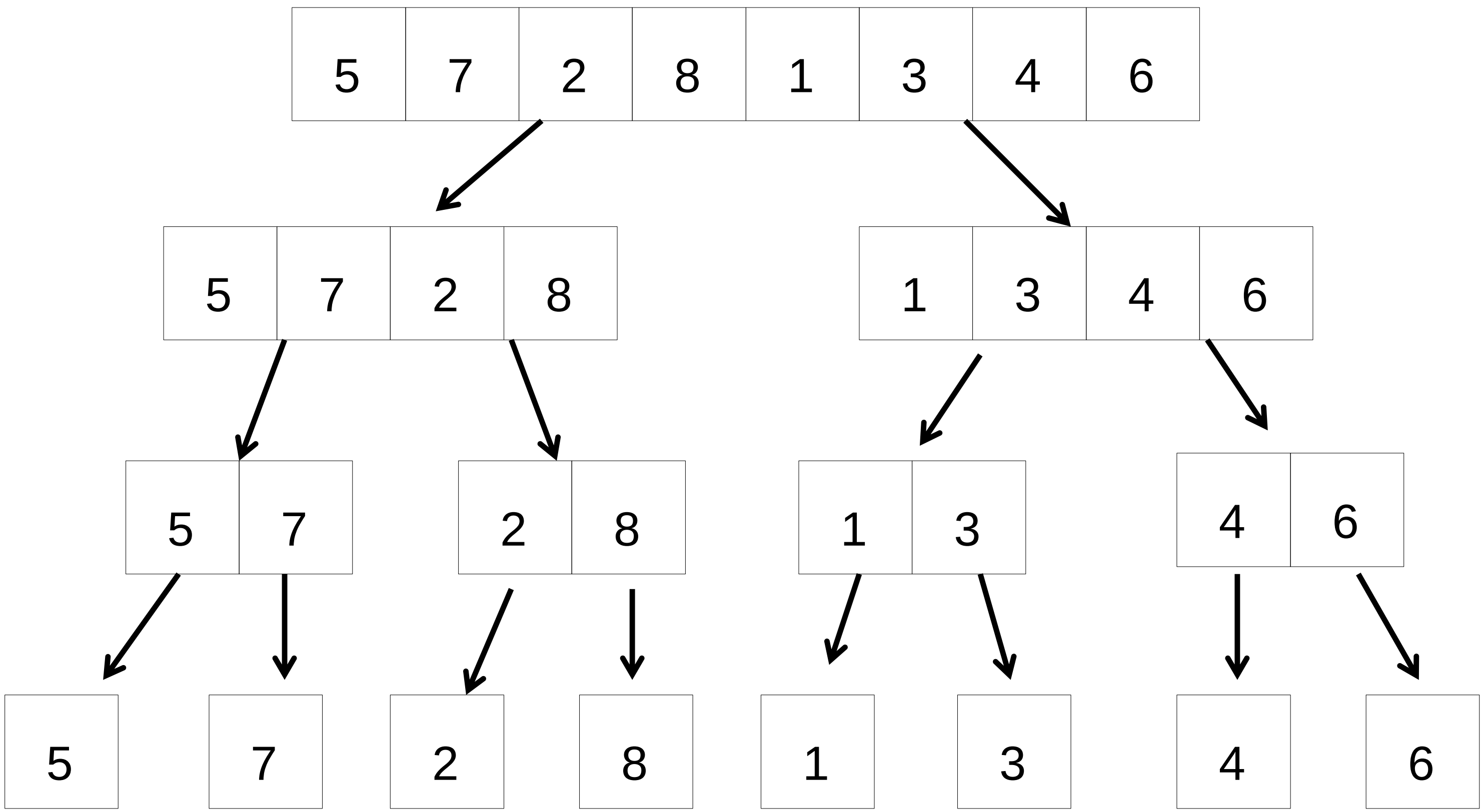
Merge Sort

```
void MergeSort(int vec[], int size) {
        int middle;
        if (size> 1) {
                middle= size/ 2;
                MergeSort(vec, mid);
                MergeSort(vec + mid, size- middle);
                Merge(vec, size);
        }
}
```

- Merge Sort is an algorithm with two recursive calls.
- Each recursive call receives half of the list until the division is not possible.
- The behavior of the **merge()** function is to sort the two halves in a complexity O(n), since each half is sorted.

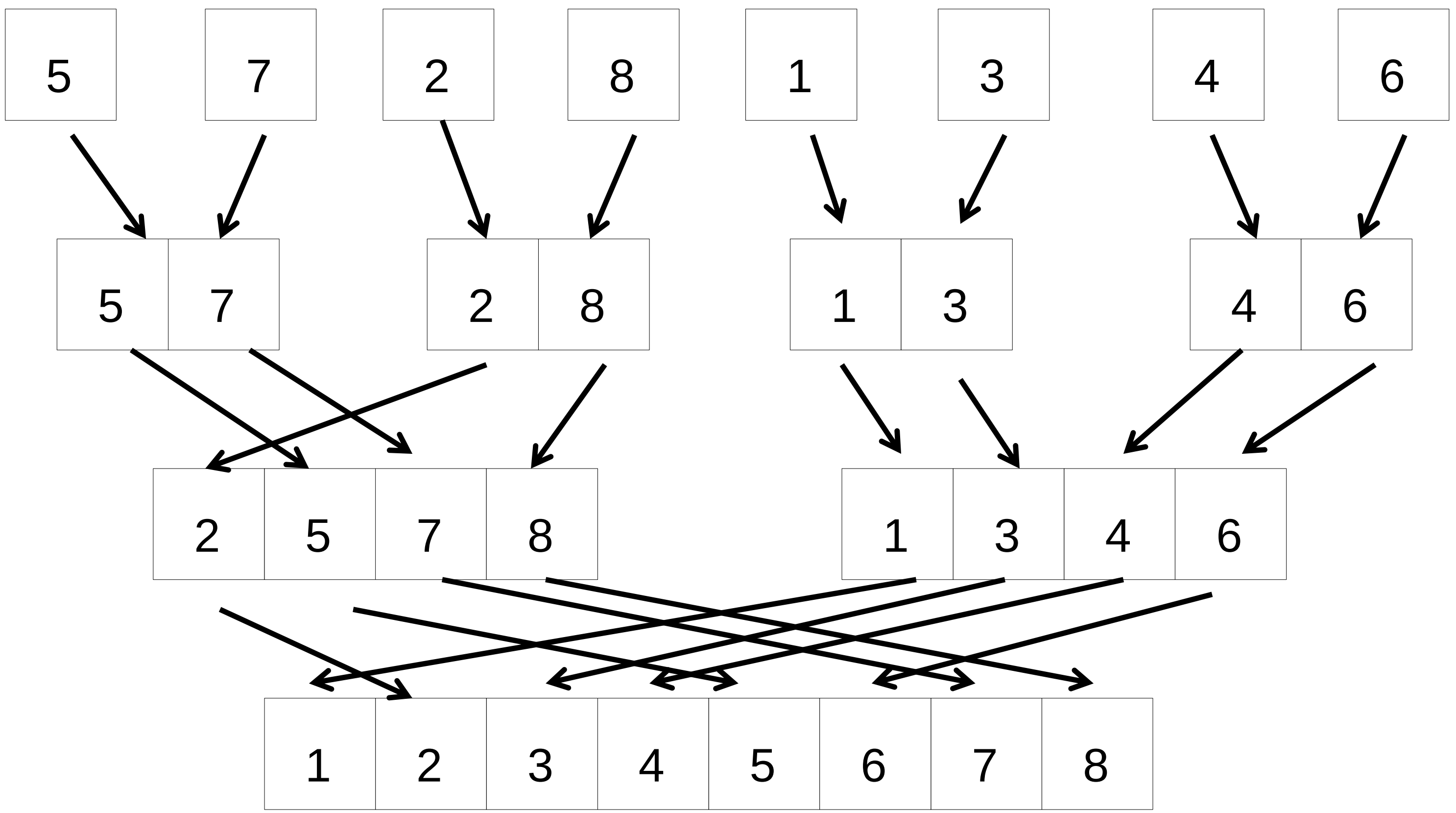
Merge Sort

Divide the list by recurrence



Merge Sort

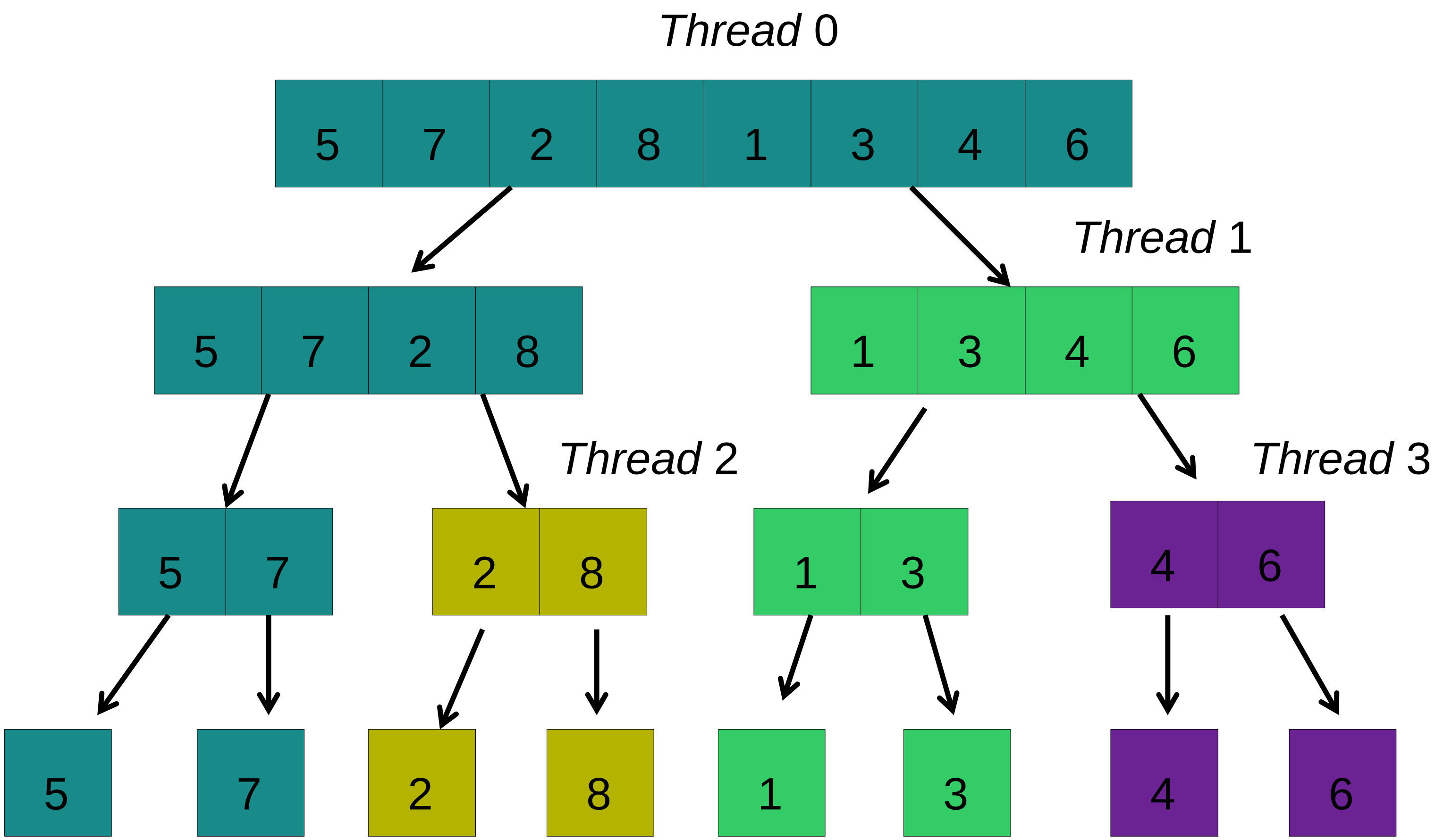
Join and sort the list



Merge Sort

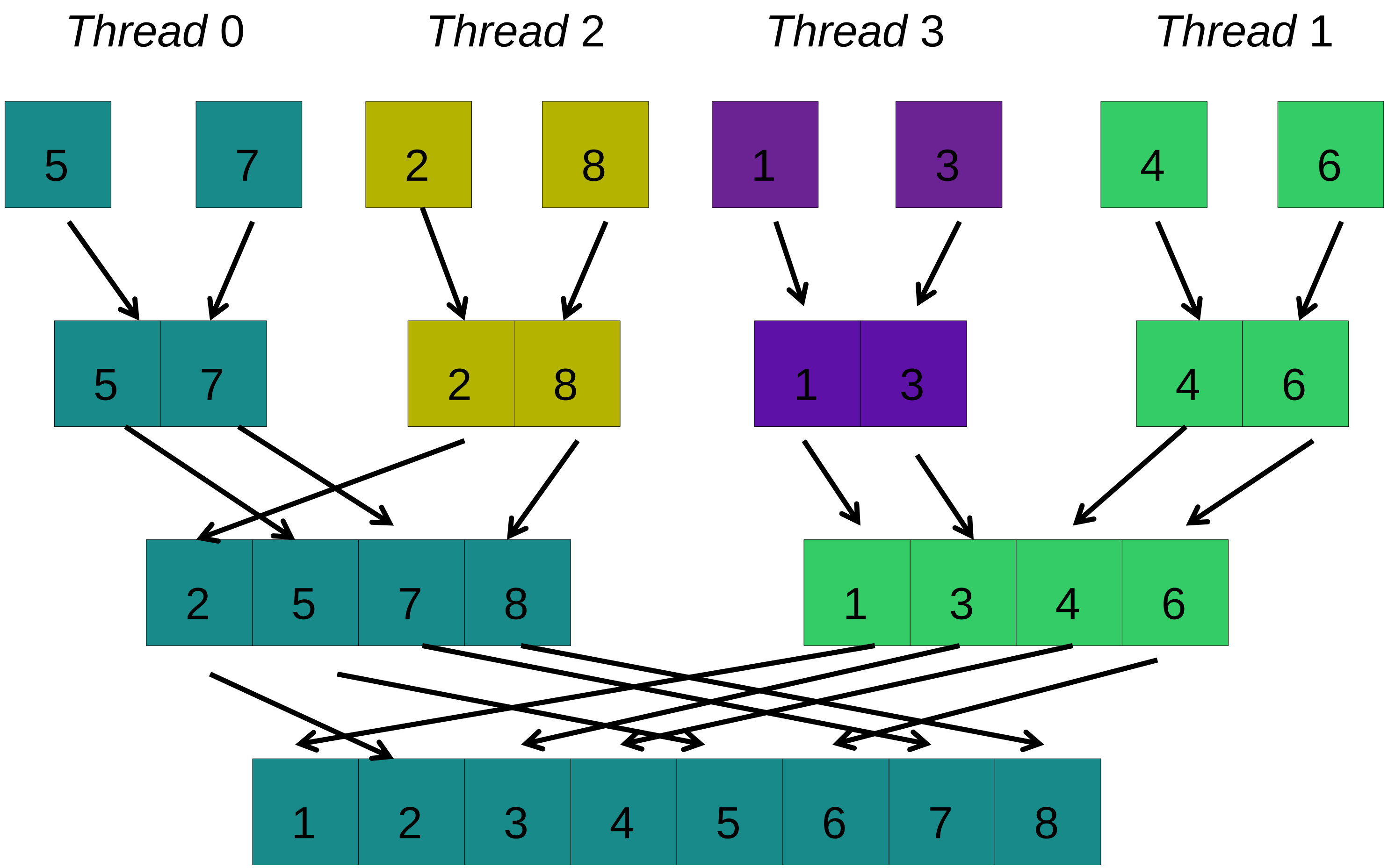
First step:
- Where is the concurrency?
- Where are the regions with independent instructions that can be executed out-of-order?

Merge Sort



Thread 0
5 7 2 8 1 3 4 6
Thread 1
5 7 2 8
1 3 4 6
Thread 2
Thread 3
5 7
2 8
1 3
4 6
5
7
2
8
1
3
4
6

Merge Sort



Thread 0
Thread 2
Thread 3
Thread 1
5
7
2
8
1
3
4
6
5 7
2 8
1 3
4 6
2 5 7 8
1 3 4 6
1 2 3 4 5 6 7 8

Merge Sort

```
void MergeSortOmp(int arr[], int size, int thread){
    int mid;
    if(size > 1) {
        mid = size / 2;
        if(thread > 1) {
            #pragma omp parallel sections
            {
                #pragma omp section
                {
                    MergeSortOmp(arr, mid, thread/2);
                }
                #pragma omp section
                {
                    MergeSortOmp(arr + mid, size - mid, thread/2);
                }
            }
        } else {
            MergeSortOmp(arr, mid, thread);
            MergeSortOmp(arr + mid, size - mid, thread);
        }
        Merge(arr, size);
    }
}
```

Main References

- Peter Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann, 2011
- OpenMP Specifications, available at https://www.openmp.org/specifications/
- Discipline notes: Parallel Programming (Graduate Program in Informatics, PUC Minas, Professor Henrique C. Freitas) and Algorithms and Data Structures II (Undergraduate Program in Computer Science, PUC Minas, Professor Max V. Machado)