PARALLELISM

Alessandro Casalino
a.casalino@cineca.it

May 23, 2023

CINECA

Outline

- Introduction
- `parallel` Construct
- Worksharing Constructs
- Scheduling Directives
- Data Clauses
- Synchronization Constructs

Introduction

Why OpenMP?

```
void add(double* x, double* y,
         double a, size_t size)
{
    for(size_t i = 0; i < size; i++)
    {
        x[i] = a * x[i] + y[i];
    }
}
```

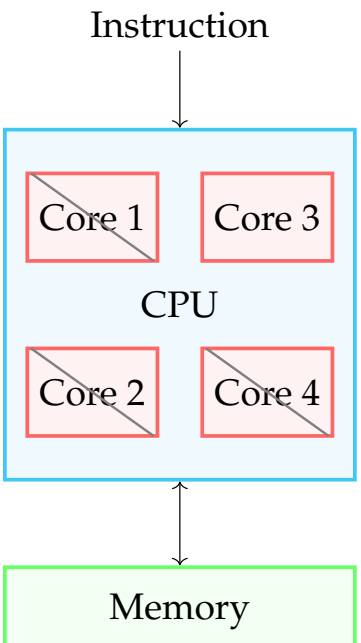
Consider this **serial** code. We want to run it on our *single* machine.

- Uses one available core (the others are idle)
- Instructions performed one by one
- No concurrent access to memory

Waste of resources!

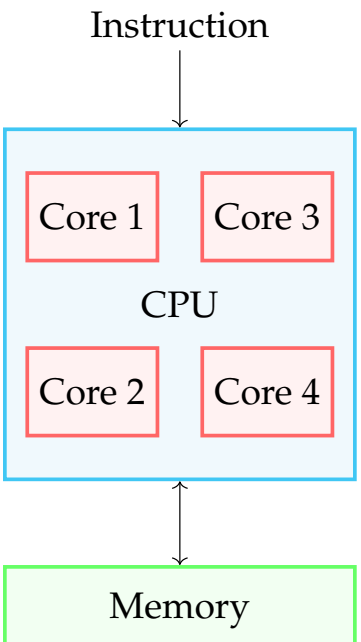
WHY OPENMP?

```
void add(double* x, double* y,
         double a, size_t size)
{
    for(size_t i = 0; i < size; i++)
    {
        x[i] = a * x[i] + y[i];
    }
}
```



Why OpenMP?

```
void add(double* x, double* y,
         double a, size_t size)
{
    // Concurrently...
    // On core 1
    for(size_t i = 0; i < size/4; i++)
    {
        x[i] = a * x[i] + y[i];
    }
    // On core 2
    for(size_t i = size/4; i < size/2; i++)
    {
        x[i] = a * x[i] + y[i];
    }
    // ...
}
```



Notation Intermezzo

Core (Hardware)

- Hardware construct
- Single independent computing unit
- Reads and execute **process** instructions

Process (Software)

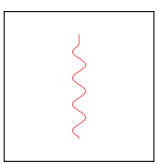
- Program under execution
- Might contain several concurrent execution flows (**threads**)
- Number of threads is dynamic during execution

Thread (Software)

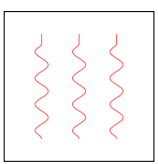
- Smallest unit of execution
- All threads have access to process memory and system resources
- Several concurrent threads might improve performance (parallelism)
- Usually one threads is mapped to a single core

Notation Intermezzo

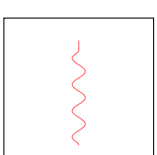
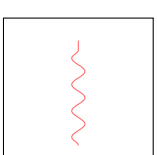
Difference between **process** and **thread**.



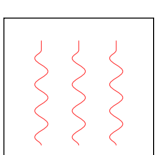
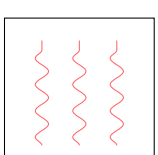
One process
One thread



One process
Multiple threads

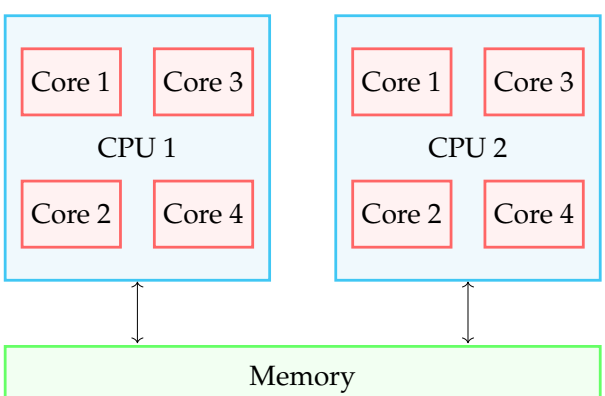
Multiple processes
One thread

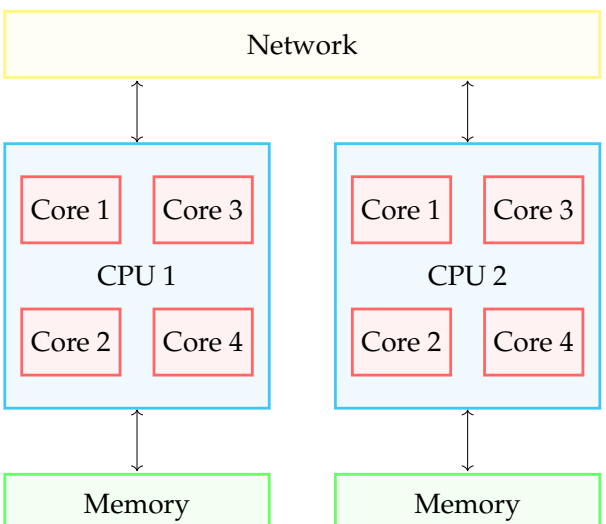
Multiple processes
Multiple threads

WHY NOT MPI?

Shared Memory

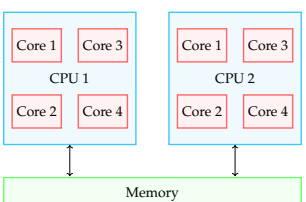


Distributed memory



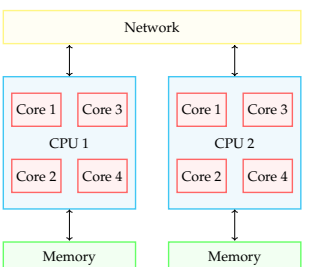
Why not MPI?

Shared Memory



- Multiple processing units
- Shared memory

Distributed memory



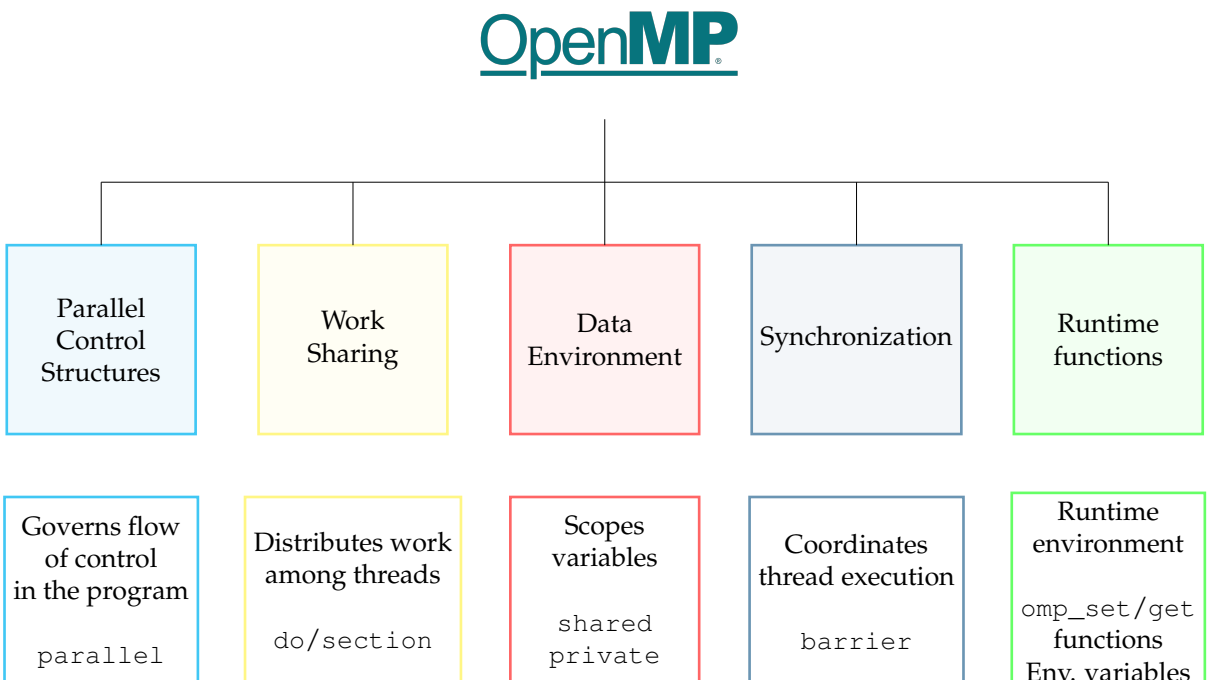
- More than one computing unit
- Each with own processing units and memory
- Interconnected via (hopefully fast) network

What can OpenMP do?

Parallelism paradigms supported by OpenMP standard.

- **Threaded parallelism** (multi-core, shared memory)
- **Vectorized execution** (SIMD)
- **Offload execution** on GPUs

What is OpenMP?



COMPILER DIRECTIVES

C/C++

```
#pragma omp construct [clauses]
{
    // Code to parallelize
}
```

Fortran

```
!$omp construct [clauses]
// Code to parallelize
!$omp end construct
```

OpenMP is **directive based**.

- Directives added to serial code
- Easy to port serial codes to multi-core CPU (and GPU)
- Compatible with C/C++ and **Fortran**

COMPILER FLAGS

Compiler flags to enable OpenMP.

Compilers	`Fortran`	`C/C++`
GNU	`gfortran *.f90 -fopenmp`	`gcc/g++ *.c/cpp -fopenmp`
Intel	`ifort *.f90 -qopenmp`	`icc/icpc *.c/cpp -qopenmp`
PGI	`pgf90 *.f90 -mp`	`pgcc *.c/cpp -mp`

Note

Pre-processor directives are ignored (without compiler errors) if the OpenMP compiler flag is missing. Thus remember to include this flag to compile with OpenMP parallism.

`parallel` construct

Example: Hello World!

```
#include <iostream>

int main()
{
    // Serial code
    std::cout << "Hello World!"
              << std::endl;
    // Serial code

    return 0;
}
```

Question

What happens if I parallelize this simple code?

Example: Hello World!

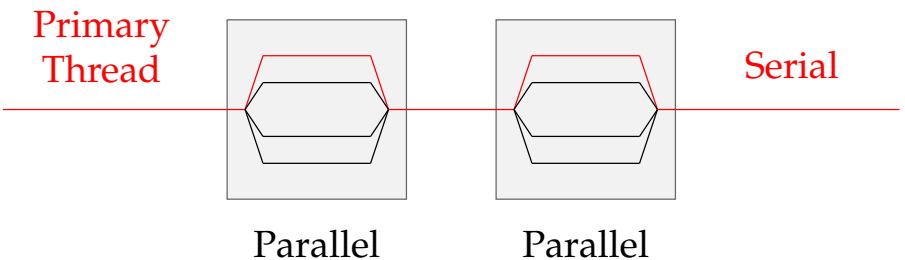
```cpp
#include <iostream>

int main()
{
    // OpenMP parallel directive
    #pragma omp parallel
    {
    std::cout << "Hello World!"
              << std::endl;
    }

    return 0;
}
```

Output

```
Hello World!
Hello World!
...
```

What's happening?

Fork-Join Parallelism



Fork:

- Primary thread spawns a team of secondary threads as needed
- Parallelism added incrementally until performance goals are met

Join:

- At the end of parallel region, thread team ends and only primary thread remains
- Primary thread execution is serial

Example: Hello World!

```
#include <iostream>

int main()
{
    // OpenMP parallel directive
    #pragma omp parallel
    {
    std::cout << "Hello World!"
              << std::endl;
    }

    return 0;
}
```

What's happening?

- Threads **redundantly** execute code in the block
- Each thread evaluates `std::cout`
- Threads are synchronized at the end of the parallel region

What about the number of times `Hello World!` is printed?

Number of threads

Set the number of threads on the **terminal**.

- For the entire terminal session
  `export OMP_NUM_THREADS=16; ./program`
- For the single execution
  `OMP_NUM_THREADS=16 ./program`

Or in the **code**.

- In the OpenMP directive

  ```
  #pragma omp parallel num_threads(16)
  ```

- With `omp` function (affects subsequent parallel regions)

  ```
  void omp_set_num_threads(int num_threads)
  ```

Example: Hello World!

```cpp
#include <iostream>

int main()
{
    // OpenMP parallel directive
    #pragma omp parallel num_threads(4)
    {
    std::cout << "Hello World!"
              << std::endl;
    }

    return 0;
}
```

Best practice

Rule of thumb: set the number of (software) threads = numbers of (hardware) cores.

Best practice

Avoid *magic numbers* in any code unless required. Prefer the `OMP_NUM_THREADS` option as bad things might happen: godbolt.org/z/K8v35dcnr.

Note

OpenMP thread number starts from 0 up to `num_threads` - 1.

Example: Hello World!

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

int main()
{
    // Note the header include above!
    omp_set_num_threads(4);

    #pragma omp parallel
    {
    std::cout << "Hello World!"
              << std::endl;
    }

    return 0;
}
```

Warning

Remember to include `omp.h` if OMP functions are used.

Example: Hello World!

Run: `OMP_NUM_THREADS=1 ./hello`

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

int main()
{
    omp_set_num_threads(2);

    // Note the num_threads clause
    #pragma omp parallel num_threads(4)
    {
    omp_set_num_threads(8);
    std::cout << "Hello World!"
              << std::endl;
    }

    return 0;
}
```

Question

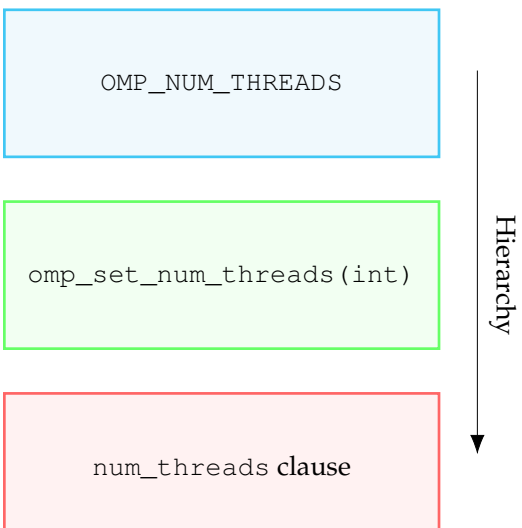
How many times `Hello World!` is printed in this case?

godbolt.org/z/xcTacrq1c

Number of threads hierarchy

Run: `OMP_NUM_THREADS=1 ./hello`

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

int main()
{
    omp_set_num_threads(2);

    // Note the num_threads clause
    #pragma omp parallel num_threads(4)
    {
    omp_set_num_threads(8);
    std::cout << "Hello World!"
              << std::endl;
    }

    return 0;
}
```



godbolt.org/z/xcTacrq1c

OpenMP functions

most commonly used subset

Name	Result type	Purpose
`omp_set_num_threads (int num_threads)`	none	number of threads to be created for subsequent parallel region
`omp_get_num_threads()`	int	number of threads in **currently executing** region
`omp_get_max_threads()`	int	maximum number of threads that can be created for a subsequent parallel region
`omp_get_thread_num()`	int	thread number of calling thread (zero based) in **currently executing** region
`omp_get_num_procs()`	int	number of processors available
`omp_get_wtime()`	double	return wall clock time in seconds since some (fixed) time in the past
`omp_get_wtick()`	double	resolution of timer in seconds

Data Race

```cpp
#include <iostream>

int main()
{

    #pragma omp parallel num_threads(4)
    {
    std::cout << "Hello World!"
              << std::endl;
    }

    return 0;
}
```

Output

```
Hello World!Hello World!

Hello World!
Hello World!
```

Warning

`std::cout` seems to be accessed by threads non-orderly!

godbolt.org/z/5zx9GcnWv

Data Race

Note

Atomic means that the operation is either completed or not. The other threads do not catch it in the middle of the operation.

Data Race

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

int main() {
    // Thread ID
    int tid, nthreads {4};
    omp_set_num_threads(nthreads);

    #pragma omp parallel
    {
    tid = omp_get_thread_num();
    nthreads=omp_get_num_threads();

    // Do some lengthy computation
    for(int i = 0; i < 1e5; i++) {};

    printf("Thread %d of %d.\n",
           tid, nthreads);
    }

    return 0;
}
```

Output

```
Thread 1 of 4.
Thread 1 of 4.
Thread 3 of 4.
Thread 3 of 4.
```

godbolt.org/z/WfW6qY9aK

Data Race

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

int main() {
    // Thread ID
    int tid, nthreads {4};
    omp_set_num_threads(nthreads);

    #pragma omp parallel private(tid)
    {
    tid = omp_get_thread_num();
    nthreads=omp_get_num_threads();

    // Do some lengthy computation
    for(int i = 0; i < 1e5; i++) {};

    printf("Thread %d of %d.\n",
           tid, nthreads);
    }

    return 0;
}
```

Possible solutions.

- `private` clause (see example)
- Define `tid` inside the parallel block

godbolt.org/z/K83q41fTv

Exercise 1

- Add a `parallel` in the correct place
- Get thread IDs in the parallel region
- Compile
- Run
- *Optional:* experiment with `OMP_NUM_THREADS`

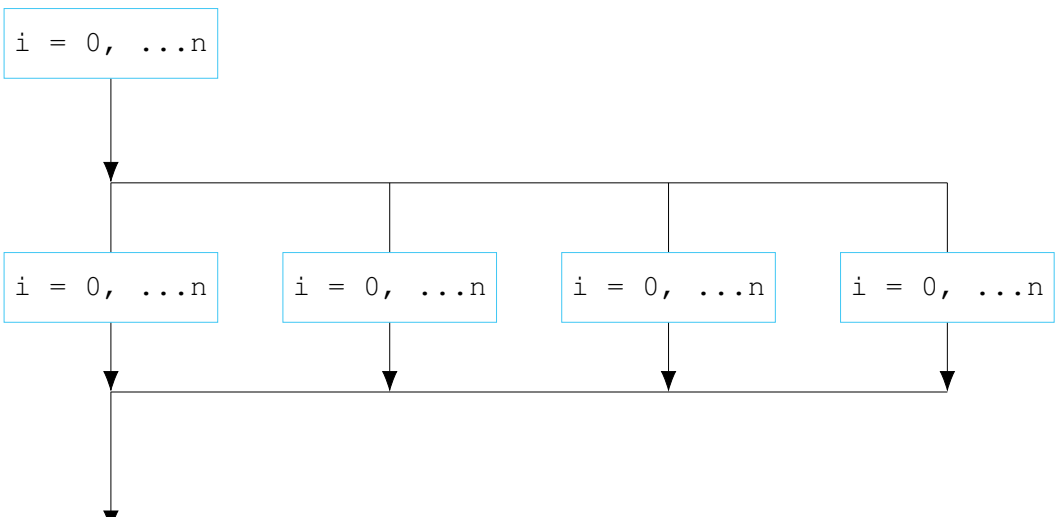
Question
Do you get what you expected?

Exercise 1 - Parallel

```
auto tick = omp_get_wtime();

#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int nthreads{omp_get_num_threads()};

    for (std::size_t i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}

auto tock = omp_get_wtime();
```

- Check the output: is it correct?
- How many times are we performing the sum on each element of `c`?

Exercise 1 - Parallel



Warning

`parallel` is executing the same code redundantly on every thread.

Worksharing constructs

Exercise 1 - Manual Worksharing

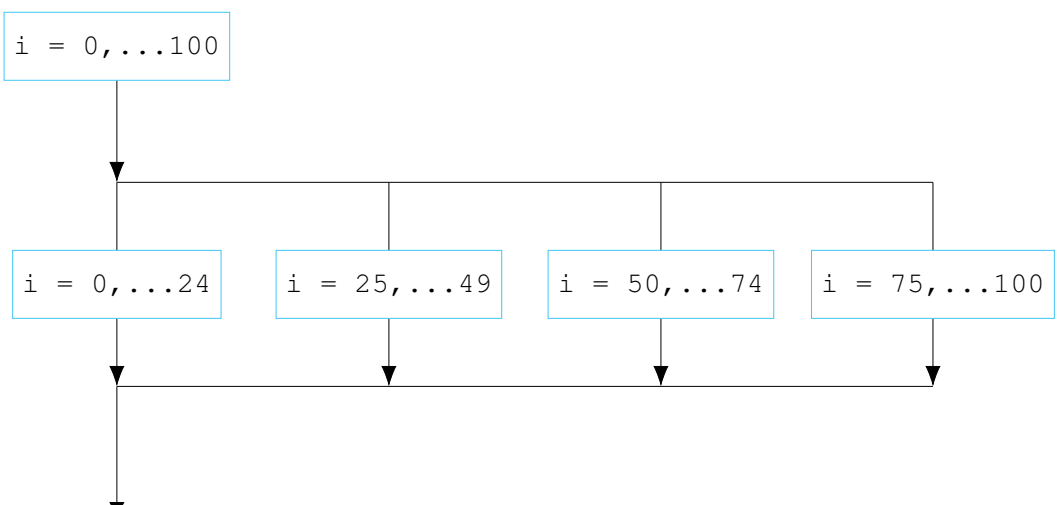
- Using the thread ID and the number of threads, subdivide the work between the threads (**worksharing**)
- Substitute the time functions with the appropriate `omp` functions
- Compile
- Run
- *Optional:* experiment with `OMP_NUM_THREADS`

Question

Do you get what you expected?

Exercise 1 - Manual Worksharing

```
for (std::size_t i = start; i <= end; i++) c[i] = a[i] + b[i];
```



Exercise 1 - Manual Worksharing

```
auto tick = omp_get_wtime();

#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    // Iteration start and end
    std::size_t start = tid * n/nthreads;
    std::size_t end = (tid+1) * n/nthreads - 1;

    for (std::size_t i = start; i <= end; i++) {
        c[i] = a[i] + b[i];
    }

}

auto tock = omp_get_wtime();
```

- Check the output: is it correct?
- How many times are we performing the sum on each element of `c`?

SPMD (Single Program Multiple Data)

Single Program Multiple Data (SPDM) is the dominant style of parallel programming, where all processors use the same program, though each has its own data.

Worksharing in SPMD

- Workshare among a **team** of threads
- Region with no internal **barriers**, i.e., do not internally wait for other threads to continue
- **Implicit** barrier at the end
- Each thread must encounter the same worksharing regions and barriers

FOR CONSTRUCT

```
// Note the presence of parallel as well
#pragma omp parallel for
for (i = 0; i < n; i++)
{
    c[i] = a[i] + b[i];
}
```

- Team of threads is formed at parallel region
- Loop iterations are split among threads
- Explicit barrier at the end of the loop

Warning

Each loop iteration must be independent of the other iterations!

Note

`private(i)` is not needed, as loop iterators are private by default.

FOR CONSTRUCT

`for` construct to distribute work among threads

```
#pragma omp parallel {
    #pragma omp for {
        for (size_t i = 0; i < n; i++)
        {
            c[i] = a[i] + b[i];
        }
    }
}
```

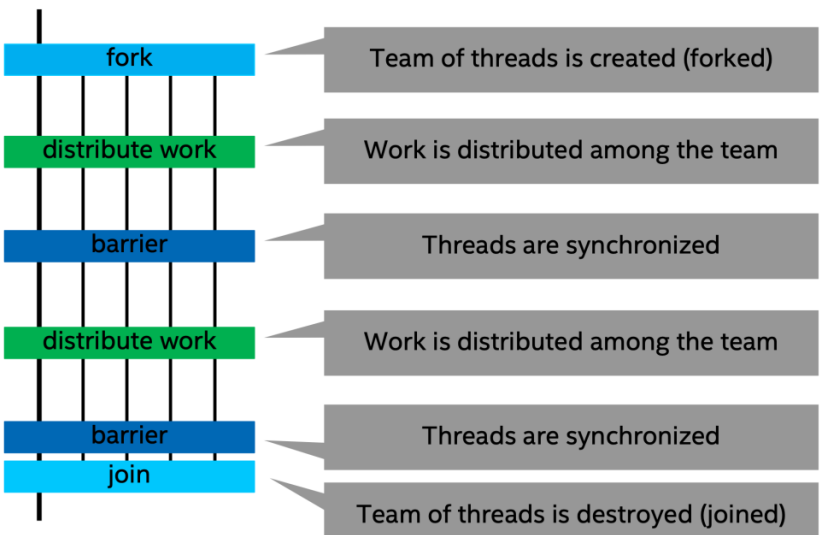
- `for` construct in `parallel` region

```
#pragma omp parallel for
for (size_t i = 0; i < n; i++)
{
    c[i] = a[i] + b[i];
}
```

- Or equivalently for just a `for`

FOR CONSTRUCT

```
#pragma omp parallel {
    #pragma omp for {
        for (i = 0; i < n; i++)
        {
            ...
        }
    }
    /* Implicit barrier */
    #pragma omp for {
        for (i = 0; i < n; i++)
        {
            ...
        }
    }
    /* Implicit barrier */
}
```



EXERCISE 1 - `FOR` CONSTRUCT

- Switch from single `parallel` to `parallel for`
- Compile
- Run
- *Optional:* experiment with `OMP_NUM_THREADS`

Question
Is the workload correctly split among the threads?

Nested loops

```
for(size_t i {0}; i < n; i++) {
    for(size_t j {0}; j < n; j ++) {
        ...
    }
}
```

What about this nested loops?

Note

Usual example: *matrix multiplication*.

$$C_{i,j} = \Sigma_{k=0}^{n} A_{i,k} B_{k,j}$$

with 3 loops (i, j, k).

Nested loops

```
#pragma omp parallel for
for(size_t i {0}; i < n; i++) {

    #pragma omp parallel for
    for(size_t j {0}; j < n; j ++) {

        ...

    }

}
```

Warning

This compiles but do not work!

- Nested parallelism is disabled in OpenMP.
- The second `pragma` is effectively ignored.
- An *inner* team of one thread only is created, i.e., each inner loop iteration is processed by one thread
- But we create more overhead in the inner loop…

NESTED LOOPS: `COLLAPSE`

```
#pragma omp parallel for collapse(2)
for(size_t i {0}; i < n; i++) {

    for(size_t j {0}; j < n; j ++) {

        ...

    }

}
```

- Loops are collapsed into one `for`
- Iterations are shared between threads
- Each thread computes its assigned portion of iteration space
- Threads synchronize and join

Note

`collpase` might introduce additional scheduling overhead and increase computational time. Time your program after collapsing loops.

EXERCISE 2

- Parallelize the code in the appropriate regions
- Try without `collapse`
- Try with `collapse(2)`
- Check the results against changes in `OMP_NUM_THREADS`
- Optional: add another loop to perform the matrix multiplication. What does it change between `collapse(2)` and `collapse(3)`?

Question

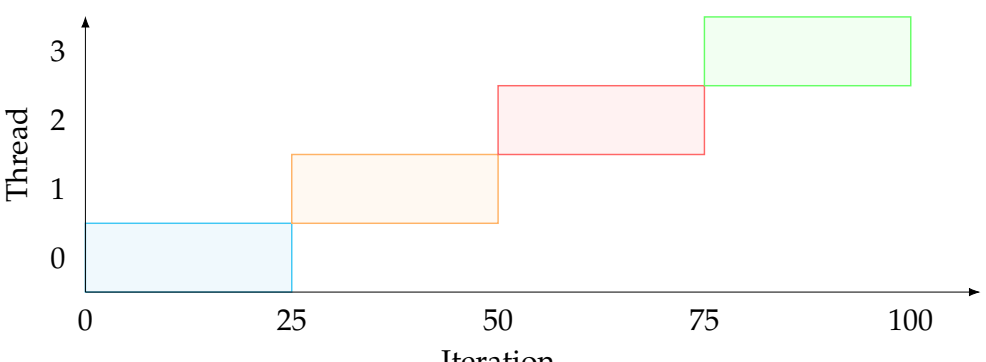
Do you get the expected speed-up after parallelizing?

Scheduling directives

SCHEDULING

How OpenMP assign the iterations to each thread?

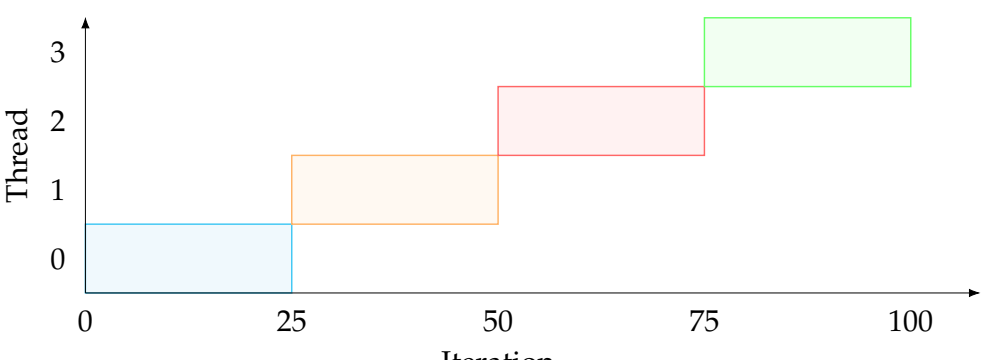
- Loop is splitted in `chunks`
- By default: `static` schedule with fixed `chunk` value



SCHEDULING: STATIC

```
#pragma omp parallel for schedule(static)
for (i = 0; i < 100; i++) {...};
```

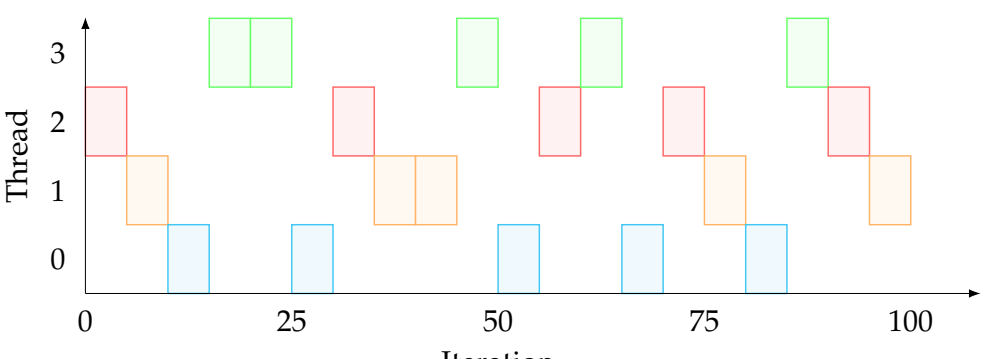
- Iterations divided into `chunks`
- Smallest overhead
- Specify chunk size with `schedule(static, 25)`
- Default `chunk` size divides the iterations equally



SCHEDULING: DYNAMIC

```
#pragma omp parallel for schedule(dynamic)
for (i = 0; i < 100; i++) {...};
```

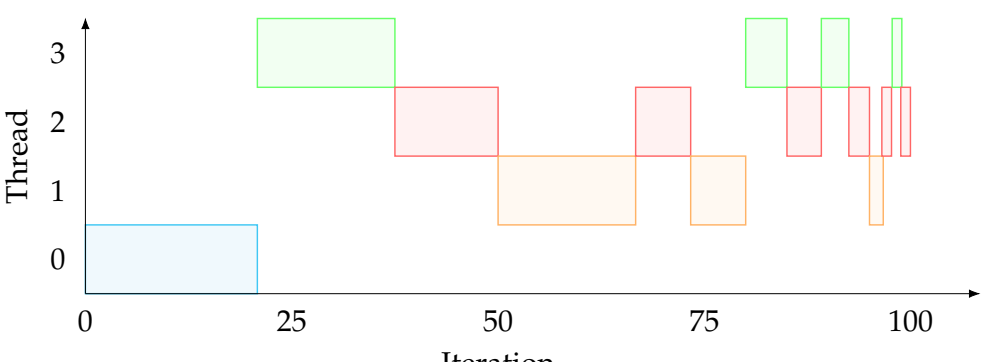
- Iterations divided into `chunks`
- Each thread requests and executes a `chunk`, until no `chunks` remains
- Useful for unbalanced workloads, e.g., when some threads complete work faster
- Default `chunk` size is 1



SCHEDULING: GUIDED

```
#pragma omp parallel for schedule(guided)
for (i = 0; i < 100; i++) {...};
```

- Similar to `dynamic`, but `chunk` size decreases in time
- Useful for unbalanced workloads, e.g., when some threads complete work faster
- But mitigates overhead of `dynamic` by starting with large `chunks`



SCHEDULING

Schedule	Overhead	Default `chunk`	Use case
`static`	Small	Equally divided by threads	Small work unbalance
`dynamic`	Big	1	Unbalanced workload
`guided`	Medium	1	Unbalanced workload

Best practice

Use `static` for generic loops to avoid overhead. Change it when the loop/architecture is known to perform better with another.

Note

`auto` schedule is also available. The compiler tries to choose the best schedule.

Exercise 3

- Parallelize the code in the appropriate regions
- Add work sharing constructs
- Experiment with `schedule` clause
- Check the results against changes in `OMP_NUM_THREADS`

Question

What is the best schedule for this work?

Question

Do you get the expected speed-up after parallelizing?

Amdahl's Law

$$\text{Speedup}\ (n) = \frac{1}{(1-p) + p/n}$$

- n: the number of workers
- $0 < p < 1$: fraction of parallelizable code

Example

- $n = 2$ processors
- $p = 0.7$ of the work is parallelizable

Theorical max Speedup $= 1.43$.

Warning

Thus sometime it might be better to revise the algorithm than blindly throw more hardware into the computation...

OpenMP Overhead/Scalability

OpenMP programs do not always scale well,
i.e., increasing OpenMP threads might not increase speedup.

- Considerable overhead involved (e.g., `schedule`)
- More threads competing for available memory bandwidth
- Cache mismanagement

Best practice

Parallelize most outer loop possible (in some cases even for small iterations)

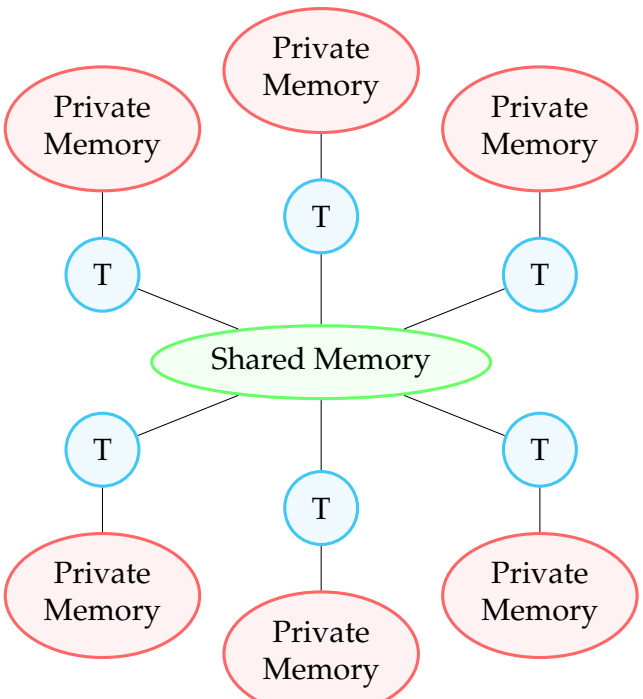
Best practice

Check the speedup against the overheads.

- Is the number of iterations large enough?
- Is the amount of work per iteration enough?

Data constructs

Memory Model



- **Shared memory**: available to all threads
- **Private memory**: available to only one thread

Warning

It is up to the programmer to choose the memory scope to variables.

SHARED CLAUSE

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

int main()
{
    // Thread ID
    int x {3};
    omp_set_num_threads(8);

    #pragma omp parallel shared(x)
    {
        auto tid = omp_get_thread_num();
        x += 1;
        printf("Thread %d has x: %d.\n",
               tid, x);
    }

    return 0;
}
```

`shared` variables have shared memory among all threads.

Output

```
Thread 0 has x:  4.
Thread 3 has x:  5.
Thread 1 has x:  6.
Thread 2 has x:  7.
```

Note

Variables are `shared` by default, i.e., the `shared` clause is redundant in this example.

godbolt.org/z/TahqMEv1E

PRIVATE CLAUSE

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

int main()
{
    // Thread ID
    int x {3};
    omp_set_num_threads(3);

    #pragma omp parallel for private(x)
    for(size_t i = 0; i < 9; i++)
    {
        auto tid = omp_get_thread_num();
        x = i;
        printf("Thread %d has x: %d.\n",
               tid, x);
    }

    printf("Final x: %d.\n", x);
    return 0;
}
```

`private` variables are **uninitialized** copies of the global ones, and visible to its thread.

Output

```
Thread 0 has x:  4.
Thread 0 has x:  0.
Thread 0 has x:  1.
Thread 0 has x:  2.
Thread 2 has x:  6.
Thread 2 has x:  7.
Thread 2 has x:  8.
Thread 1 has x:  3.
Thread 1 has x:  4.
Thread 1 has x:  5.
Final x:  3.
```

godbolt.org/z/ncYG1n88z

PRIVATE CLAUSE

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

int main()
{
    // Thread ID
    int x {3};
    omp_set_num_threads(3);

    #pragma omp parallel for private(x)
    for(size_t i = 0; i < 9; i++)
    {
        auto tid = omp_get_thread_num();
        x += i; // Instead of =
        printf("Thread %d has x: %d.\n",
               tid, x);
    }

    printf("Final x: %d.\n", x);
    return 0;
}
```

Warning

Be careful about the initialization!

Output

```
Thread 1 has x:  5.
Thread 1 has x:  9.
Thread 1 has x:  14.
Thread 2 has x:  6.
Thread 2 has x:  13.
Thread 2 has x:  21.
Thread 0 has x:  -608901888.
Thread 0 has x:  -608901887.
Thread 0 has x:  -608901885.
Final x:  3.
```

godbolt.org/z/xvb4bfs45

FIRSTPRIVATE CLAUSE

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

int main()
{
    // Thread ID
    int x {3};
    omp_set_num_threads(3);

    #pragma omp parallel for \
            firstprivate(x)
    for(size_t i = 0; i < 3; i++) {
        auto tid = omp_get_thread_num();
        printf("Thread %d has x: %d.\n",
               tid, x);
        x = i;
        printf("Thread %d has x: %d.\n",
               tid, x);
    }
    printf("Final x: %d.\n", x);
    return 0;
}
```

`firstprivate` variables are copies of the global ones, visible to its thread, and initialized to the global variable value.

Output

```
Thread 0 has x:  3.
Thread 0 has x:  0.
Thread 2 has x:  3.
Thread 2 has x:  2.
Thread 1 has x:  3.
Thread 1 has x:  1.
Final x:  3.
```

Best practice

Prefer `firstprivate` to avoid unwanted uninitialized variables.

godbolt.org/z/3jv6K84br

LASTPRIVATE CLAUSE

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

int main()
{
    // Thread ID
    int x {3};
    omp_set_num_threads(3);

    #pragma omp parallel for lastprivate(x)
    for(size_t i = 0; i < 3; i++)
    {
        auto tid = omp_get_thread_num();
        x = i;
        printf("Thread %d has x: %d.\n",
               tid, x);
    }

    printf("Final x: %d.\n", x);
    return 0;
}
```

`lastprivate` variables are **uninitialized** copies of the global ones, visible to its thread. Once the parallel region is left, it is copied back to the value of the last iteration.

Output

```
Thread 0 has x:  0.
Thread 1 has x:  1.
Thread 2 has x:  2.
Final x:  2.
```

godbolt.org/z/MeMa757x6

Data clauses

Name	Sharing Policy	Initialization	Outside Value
`shared`	All threads	Global value	Last modified
`private`	One thread	No	Pre-parallel region
`firstprivate`	One thread	Global value	Pre-parallel region
`lastprivate`	One thread	No	Last iteration

Best practice

Prefer `firstprivate` over `private` to avoid uninitialized variable issues.

Note

Default data sharing policy:

- `shared` if defined outside of the parallel region
- `private` if defined inside the parallel region or loop iterators when using the `for` construct.

DEFAULT CLAUSE

```
// Some variables with any datatype T
T x, y;

// All variables are shared
#pragma omp parallel default(shared) \
                     private(x)

// All variables are private
#pragma omp parallel default(private) \
                     shared(y)

// Force to specify explicitly the
// data sharing policy
#pragma omp parallel default(none) \
                     private(x)     \
                     shared(y)
```

`default` changes the default data policy behaviour.

Warning

Using `none`, the programmer is forced to specify the data sharing policy for all variables used in the parallel region.

Best practice

Always use `default(none)` and explicity specify the data sharing policy for all variables used in the parallel region. [clang-tidy]

Synchronization Constructs

Example

```cpp
#include <iostream>
#include <omp.h>

int main()
{
    int sum {0}, n {10000};

    auto start = omp_get_wtime();
    for(size_t i = 1; i < n; i++) {
        sum += i;
    }
    auto end = omp_get_wtime();

    std::cout << "Sum: " << sum
              << " (compare with "
              << n*(n-1)/2 << " )."
              << std::endl;
    std::cout << "Runtime: "
              << end-start << std::endl;
    return 0;
}
```

Output

```
Sum:  49995000 (compare with
49995000 ).
Runtime:  4.1008e-05
```

Question

Can you parallelize this loop?

godbolt.org/z/TqaM1nczG

Example

```cpp
#include <iostream>
#include <omp.h>

int main()
{
    int sum {0}, n {10000};

    auto start = omp_get_wtime();
    #pragma omp parallel for
    for(size_t i = 1; i < n; i++) {
        sum += i;
    }
    auto end = omp_get_wtime();

    std::cout << "Sum: " << sum
              << " (compare with "
              << n*(n-1)/2 << " )."
              << std::endl;
    std::cout << "Runtime: "
              << end-start << std::endl;
    return 0;
}
```

Output

```
Sum:  36949316 (compare with
49995000 ).
Runtime:  9.6305e-05
```

Warning

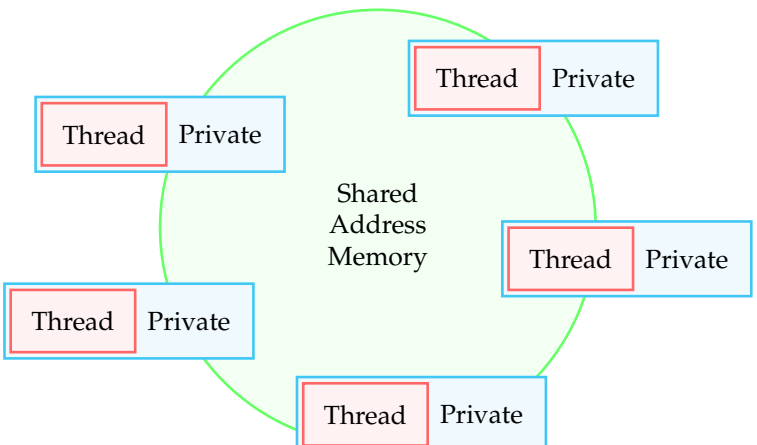
Again: **data race**!
But in this case, `private` is not the best choice...

Question

What happens using `clang` instead?

godbolt.org/z/MaMEoq6Kh

Data Race



- **Data race** occurs when two or more threads access the same memory location
- **Synchronization** to assure legal order of operations

Warning

It is up to the programmer to choose the correct synchronization method.

Data Race Solutions

Mutual Exclusion (Mutex)

Define block of code executed by one thread only at a time.

Constructs

- `critical`
- `atomic`
- `barrier`

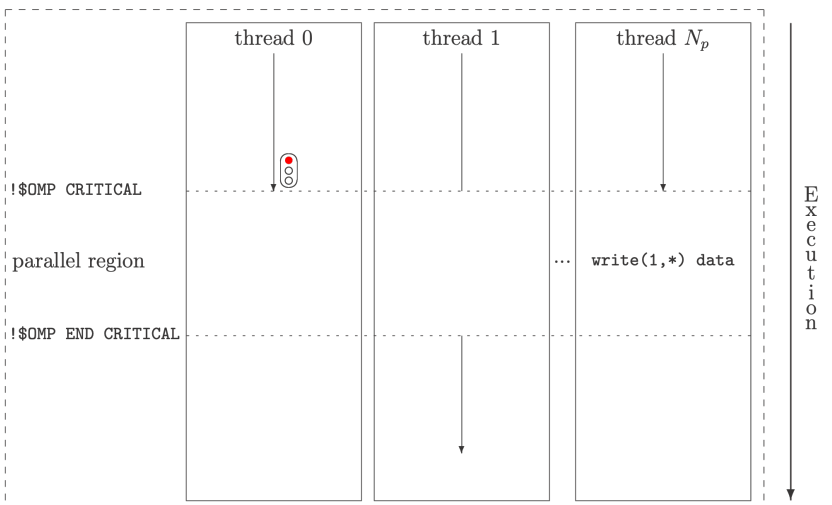
Reduction

Perform global operation on quantity from different threads.

Operations

- Sum, subtraction, multiplication
- Logical operations: or, and, ...
- Maximum, minimum

CRITICAL CONSTRUCT



`critical` prevents multiple threads from accessing a section of the code at the same time.

- **Pro**: prevents data race
- **Con**: other threads wait their turn in idle status

CRITICAL CONSTRUCT

```cpp
int sum {0}, n {10000};

auto start = omp_get_wtime();

#pragma omp parallel for
for(size_t i = 1; i < n; i++) {
    #pragma omp critical
    sum += i;
}

auto end = omp_get_wtime();
```

`critical` prevents multiple threads from accessing a section of the code at the same time.

Output

```
Sum:  49995000 (compare with
49995000 ).
Runtime:  0.000228857
```

godbolt.org/z/e8aq3MWj7

ATOMIC CONSTRUCT

```
int sum {0}, n {10000};

auto start = omp_get_wtime();

#pragma omp parallel for
for(size_t i = 1; i < n; i++) {
    #pragma omp atomic
    sum += i;
}

auto end = omp_get_wtime();
```

`atomic` guarantees mutually exclusive access to a specific memory location (variable).

Output

```
Sum:  49995000 (compare with
49995000 ).
Runtime:  0.000147173
```

godbolt.org/z/zq4b7fKr3

`atomic` CONSTRUCT LIMITATIONS

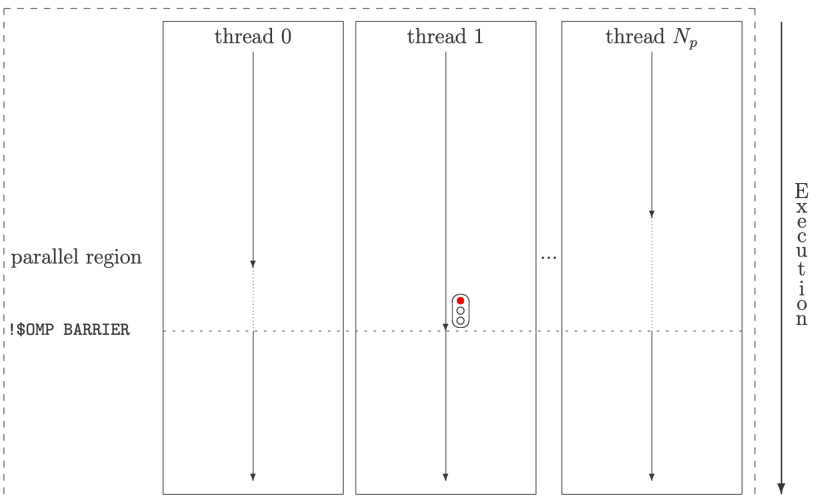
`atomic` guarantees exclusive access to a variable for the following **operations**.

- Read: `v = x`
- Write: `x = v`
- Update: `x++`, `x-`, `-x`, `+=`, `*=` ...
- Capture: `v = x++`, ...

Limitations

- `x` and `v` must be scalar
- No operator overload
- No complex expressions

BARRIER CONSTRUCT



`barrier` prevents threads from continuing the execution until all threads reach the it.

- **Pro**: prevents data race
- **Con**: other threads wait in idle status

Best practice

Use `barrier` as little as possible! If you have too many barriers it might be good to consider alternatives or rethink the algorithm.

BARRIER CONSTRUCT

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

int main()
{
    #pragma omp parallel
    {
        auto tid {omp_get_thread_num()};
        printf("Thread %d here \n", tid);

        #pragma omp barrier
        printf("Thread %d passed the "
        "barrier \n", tid);
    }
    return 0;
}
```

`barrier` prevents threads from continuing the execution until all threads reach the it.

Output without `barrier`

```
Thread 0 here
Thread 0 passed the barrier
Thread 1 here
Thread 1 passed the barrier
```

Output with `barrier`

```
Thread 0 here
Thread 1 here
Thread 0 passed the barrier
Thread 1 passed the barrier
```

godbolt.org/z/cYrrYYE84

Data Race Solutions

Mutual Exclusion (Mutex)

Define block of code executed by one thread only at a time.

Constructs

- `critical`
- `atomic`
- `barrier`

Reduction

Perform global operation on quantity from different threads.

Operations

- Sum, subtraction, multiplication
- Logical operations: or, and, ...
- Maximum, minimum

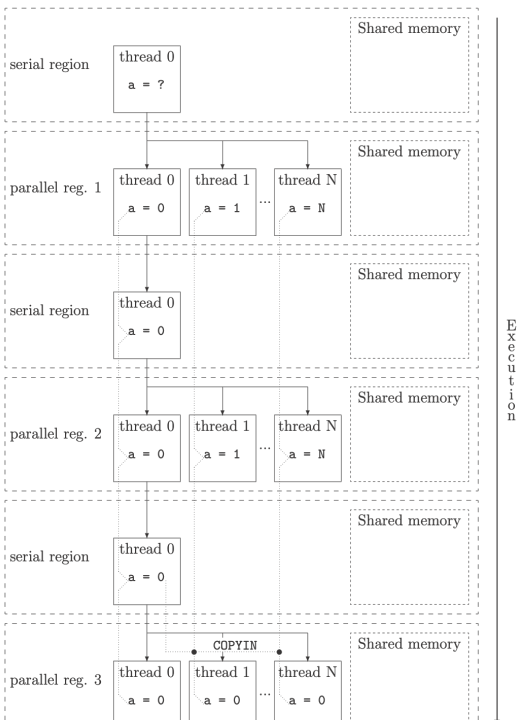
Example

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

int main()
{
    int sum {0}, n {10000};

    auto start = omp_get_wtime();
    #pragma omp parallel for
    for(size_t i = 1; i < n; i++) {
        sum += i;
    }
    auto end = omp_get_wtime();

    std::cout << "Sum: " << sum
              << " (compare with "
              << n*(n-1)/2 << " )."
              << std::endl;
    std::cout << "Runtime: "
              << end-start << std::endl;
    return 0;
}
```

We want a way to gather contributions of the sum from different threads.

- **Manual** . . . No . . .
- `reduction` clause

REDUCTION CLAUSE



`reduction` combines values on a single accumulation variable.

Syntax: `reduction(operation:var)`

Operations

- `+`
- `-`
- `*`
- `.and`
- `.or`
- `.eqv`
- `.neqv`
- `.max`
- `.min`
- `.iand`
- `.ior`
- `.ieor`

REDUCTION CLAUSE

```cpp
#include <iostream>
#include <omp.h>

int main()
{
    int sum {0}, n {10000};

    auto start = omp_get_wtime();
    #pragma omp parallel for reduction(+:sum)
    for(size_t i = 1; i < n; i++) {
        sum += i;
    }
    auto end = omp_get_wtime();

    std::cout << "Sum: " << sum
              << " (compare with "
              << n*(n-1)/2 << " )."
              << std::endl;
    std::cout << "Runtime: "
              << end-start << std::endl;
    return 0;
}
```

Output

```
Sum:  49995000 (compare with
49995000 ).
Runtime:  7.1988e-05
```

godbolt.org/z/45xKrsqox

NOWAIT CLAUSE

```
#pragma omp parallel
{
    #pragma omp for
    for(size_t i=0; i<10; i++){
        c();
    }
    d();
}

#pragma omp parallel
{
    #pragma omp for nowait
    for(size_t i=0; i<10; i++){
        c();
    }
    d();
}
```

- Without `nowait`, threads wait `for` completion before evaluating `d()`
  Example: godbolt.org/z/qbadoqG19
- With `nowait`, threads can evaluate `d()` as soon as they exit the `for`
  Example: godbolt.org/z/8hs4fhaTd

Exercise 4

Computation of π using

$$\pi = \int_0^1 \frac{4}{1+x^2}\, dx \cong \sum_{i=0}^{N} \frac{4}{1+x^2}\, \Delta x\,.$$

- Parallelize the code in the appropriate regions
- Try to add `critical` and `atomic` to solve the data race
- Then try with `reduction`
- Run and time the code
- Check the results against changes in `OMP_NUM_THREADS`

EXERCISE 4

Implementation	Runtime (s)
Serial	0.14
`critical`	10.60
`atomic`	8.34
`reduction`	0.04

Best practice

Use `reduction` when possible, especially with SPMD algorithms.

Explicit Worksharing constructs

MPMD (Multiple Program Multiple Data)

Multiple Program Multiple Data (SPDM) is a style of parallel programming, where multiple autonomous processors simultaneously operating at least two independent programs.

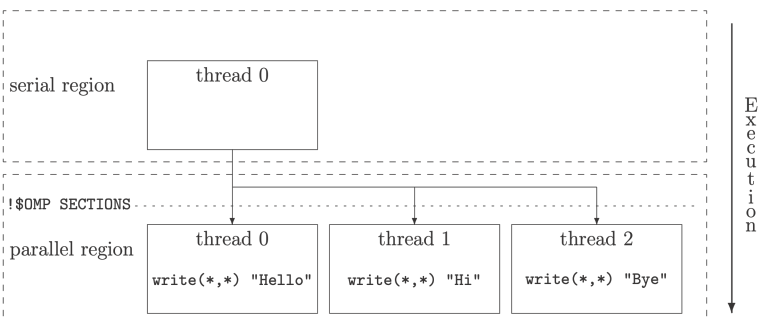
Examples

- Pre-determined independent work units
- Different functions in parallel
- Pipeline to overlap I/O with computations

SECTION CONSTRUCT

- `section` creates different tasks assigned to different threads in the parallel region.
  Examples: godbolt.org/z/6fxfde7z4 and godbolt.org/z/vz7ffb1oo

```
#pragma omp parallel sections
{
    #pragma omp section
    printf("Hello\n");
    #pragma omp section
    printf("Hi\n");
    #pragma omp section
    printf("Bye\n");
}
```



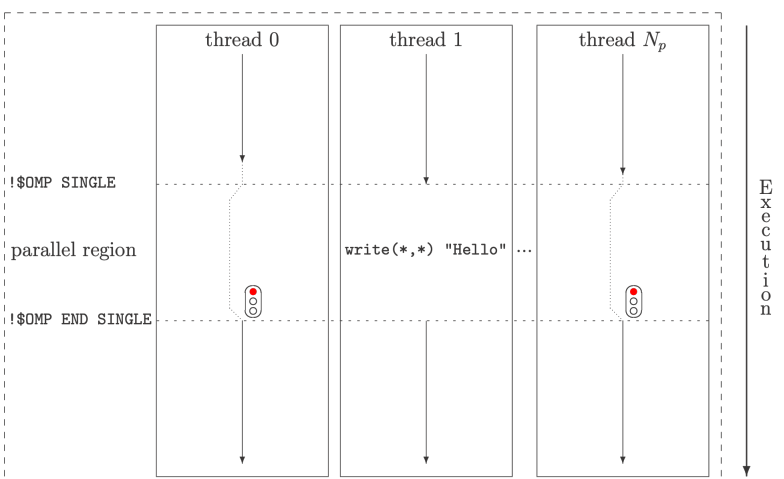
Warning

The variables defined in the parallel region are not available in the `section`.
Example: godbolt.org/z/en9jr6acv

SINGLE CONSTRUCT

- `single` assigns a region exclusively to the first available thread. There is an **implicit barrier**.
  Example: godbolt.org/z/ndhcGYba3

```
#pragma omp parallel
{
    #pragma omp single
    printf("Hello\n");
}
```



SINGLE CONSTRUCT

```cpp
#include <cstdio>
#include <omp.h>

int main()
{
#pragma omp parallel num_threads(4)
{
    auto tid = omp_get_thread_num();

    #pragma omp single
    printf("Hello from thread %d\n",
           tid);
    #pragma omp single
    printf("Bye from thread %d\n",
           tid);
    #pragma omp single nowait
    printf("Bye bye from thread %d\n",
           tid);
}
return 0;
}
```

Some outputs

Output

```
Hello from thread 1
Bye from thread 1
Bye bye from thread 2
```

Output

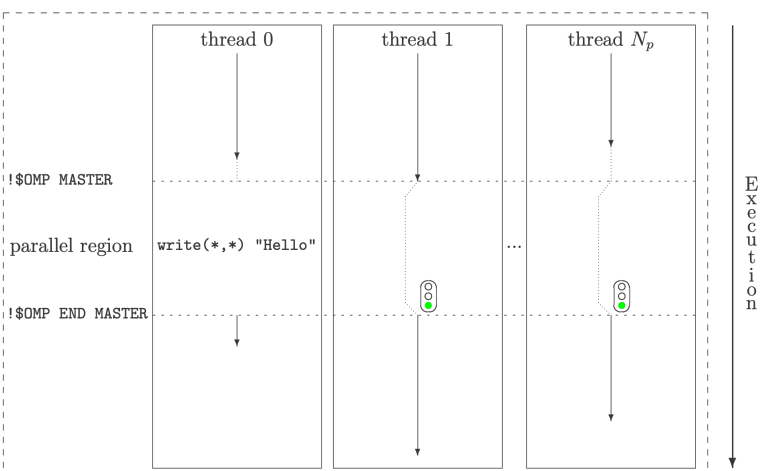
```
Hello from thread 3
Bye from thread 0
Bye bye from thread 3
```

godbolt.org/z/s6evTjeoe

MASTER CONSTRUCT

- `mater` assigns a region exclusively to the master thread. There is **no implicit barrier**. Example: godbolt.org/z/aefz6Tnba

```
#pragma omp parallel
{
    #pragma omp master
    printf("Hello\n");
}
```



Explicit Work-sharing constructs

Name	Implicit barrier	Parallel region variables	Thread executing
`section`	Yes	No	First available
`single`	Yes	Yes	First available
`master`	No	Yes	Master

Take-Home Messages

TAKE-HOME MESSAGES

- **OpenMP**: to parallelize on shared memory systems (or hybrid with `MPI`)
- **OpenMP works with threads**: master thread is forked in threads when a parallel region is reached; the control is given back to the master at the end of the parallel region
- **OpenMP** consists in directives (code), run-time routines (in `omp.h`) and environmental variables (shell)

Take-Home Messages

Constructs/Clauses

- To create teams of threads
  - `parallel`
- To share work among threads
  - `for` (SPMD)
  - `section` (MPMD)
  - `single` (MPMD)
  - `master` (MPMD)
- To prevent conflicts
  - `critical`
  - `atomic`
  - `barrier`
- Data environment clauses
  - `private`
  - `firstprivate`
  - `lastprivate`
  - `reduction`

API functions

- To manage threads
  - `omp_set_num_threads(int num_threads)`
  - `omp_get_num_threads()`
  - `omp_get_thread_num()`
  - `omp_ge_num_procs()`
- To time the code
  - `omp_get_wtime()`
  - `omp_get_wtick()`

Take-Home Messages

Best practice

Prefer `OpenMP` over `MPI` on shared memory systems: the overhead of memory transfer in `MPI` is huge. Or an hybrid approach for both shared and distributed memory systems.

Warning

Be aware of the overheads of `OpenMP` (scheduling, barriers, ...). Always time (or better profile) your code.

Take-Home Messages: C++

Warning

Be careful when using `std::cout`, `std:cin` or similar, due to data race. This is another reason a good debugger might help!

Warning

Check if some advanced functionalities of `C++` are compatible with the constructs used. For instance, avoid brace initialization for the internal loop iterator variables, e.g. **avoid**

```
#pragma omp parallel for
for(size_t i {0}; i < N; i++) { ... };
```

Resources

- **Essential**: Official OpenMP Reference Guides
- Tim Mattson's OpenMP Tutorial
- OpenMP Compilers and Tools Official List
- And obviously: Stackoverflow (with OpenMP tag)

Thank you!