

# OpenMP

## Shared-Memory Parallel Programming

Orian Louant

# Motivations for Parallel Computing

Most applications today are parallel applications

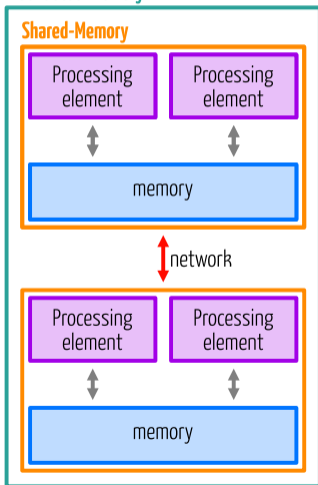
- in the years 2000's the CPU manufacturers have run out of room for boosting CPU performance
- instead of driving clock speeds and straight-line instruction throughput higher, they turn to hyperthreading and multicore architectures

In HPC in particular, it's crucial to be able to run your application in parallel

- in HPC, recent high-end CPUs have high core count: 36-64 cores
- high core count but lower clock, single core performance of an HPC CPU can be worse than your laptop CPU with turbo boost

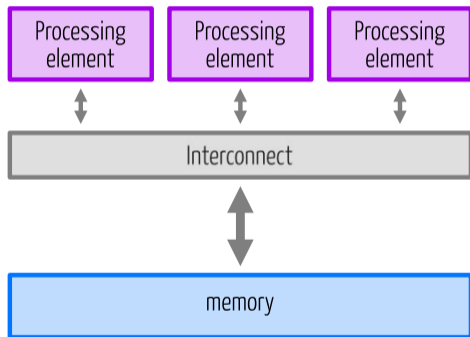
# Types of Parallel Systems

## Distributed-memory



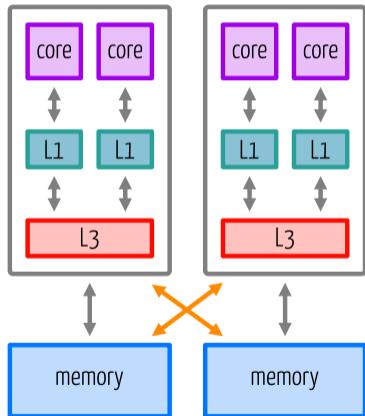
- multiple compute nodes: distributed memory
- in HPC the dominant model for distributed memory programming is MPI, a standard that was introduced by the MPI Forum in May 1994 and updated in June 1995, last version 4.0 (2021).
- single compute node: shared memory
- in HPC the dominant model for shared-memory programming is OpenMP, a standard that was introduced in 1997 (Fortran) and 1998 (C/C++), last version is 5.1 (2020)

# Shared Memory Systems



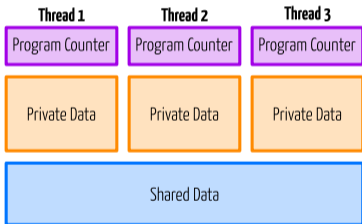
- OpenMP use threads to parallelize applications for shared memory systems
- the system consists of processing elements (CPUs, cores, GPUs) and memory
- the processing elements have access the entire memory of the system via an interconnect
- every processing elements can read and write the memory (single address space)

## Shared Memory Systems in the Real World



- the memory may be split into chunks leading to some of the core being closer (faster access) to a chunk of the memory
- different levels of cache may exist, some shared between the cores and other private to the core
- all the cores have access to the entire memory of the system but the hardware complexity may have an impact on performance

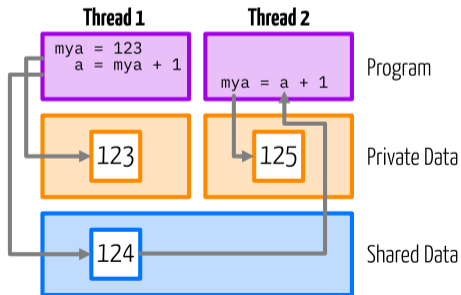
# Multithreaded Application



- thread parallelism is one of the options for shared memory systems
- all the threads can access the shared memory
- threads can have private data, which can only be accessed by the thread owning the data
- the threads have their own program counter and thus, can follow different control flow

# Thread Communication

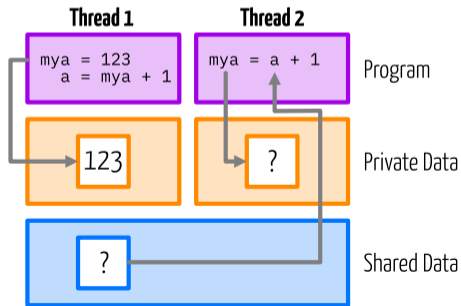
- most parallel applications require some form of communication
- in the shared memory model, threads communicate by reading and writing shared data



- thread 1 write to a shared variable
- thread 2 read the value of the shared variable

# The Synchronization Problem

- reading and writing shared data with threads executing asynchronously
- in the shared-memory model, we need to ensure that read and write operations occur in the correct order



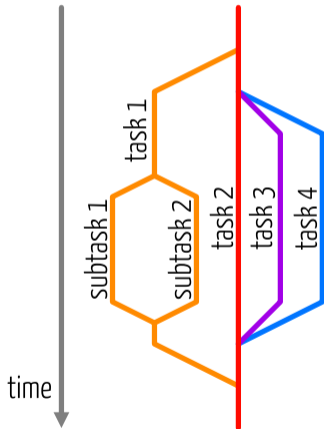
- thread 1 and thread 2 execute instructions independently of each other
- thread 2 read the value of the shared variable before thread 1 write the variable



# Data Race

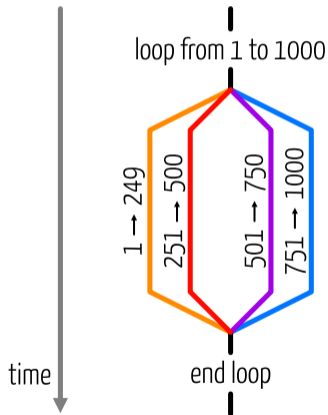
- a data race occur when one or more threads in a single process access the same memory location concurrently, and at least one of the accesses is a write
- we need synchronization to ensure proper time ordering of the memory accesses
- the lack of proper synchronization lead to non-deterministic behaviour

## Parallelism: Tasks



- task parallelism consists in executing different operations in parallel
- in this model, each thread is responsible for the execution of one task
- tasks may have sub-tasks that also execute in parallel
- writing a task parallel application consists in identifying independent computations to be assigned to the tasks

## Parallelism: Loop Parallelism



- in scientific applications, the main source of parallelism is loops
- if the iterations of a loop are independent then the loop become embarrassingly parallel
- each thread work on a portion of the iterations

# Shared-Memory Parallel Programming: What We Need

In order to write a shared-memory parallel application we need a way to

- run the program in parallel, to create threads
- specify which data is shared among the threads and which data is private to the thread
- a way to synchronize the threads, avoiding data race
- a way to create tasks and enable loop parallelism

All these elements are provided by OpenMP

## What is OpenMP?

- OpenMP is a shared-memory application programming interface
- OpenMP hides the low-level details and allows the programmer to describe the parallel code with high-level constructs: compiler directives

OpenMP consist of

- compiler directive
- runtime library routines
- environment variables

## OpenMP is using directives

Directives are programming language constructs that specifies how a compiler should process its input

An OpenMP program is the combination of

- a base language (C, C++ or Fortran)
- annotations with OpenMP directives

# Anatomy of an OpenMP directive

## OpenMP directive in C/C++

```
#pragma omp construct [clauses]
```

 Tells the compiler that it is a directive

## OpenMP directive in Fortran


```
!$omp construct [clauses]
```

# Anatomy of an OpenMP directive

## OpenMP directive in C/C++

`#pragma omp construct [clauses]`

Indicates that it is as an OpenMP directive



## OpenMP directive in Fortran

`!$omp construct [clauses]`



# Anatomy of an OpenMP directive

## OpenMP directive in C/C++

```
#pragma omp construct [clauses]
```




Give instruction on what to do

## OpenMP directive in Fortran

```
!$omp construct [clauses]
```

# Anatomy of an OpenMP directive

## OpenMP directive in C/C++

Additional options (optional) 

```
#pragma omp construct [clauses]
```

## OpenMP directive in Fortran

```
!$omp construct [clauses]
```

## The Advantages of Using directive

- does not modify the serial implementation, you can still compile and run the program as a serial code
- they can be added incrementally allowing a gradual parallelization
- directives hide the actual parallelization work from the programmer, the compiler replaces the directives by the appropriate calls to the OpenMP runtime and library
- easier to maintain

# Going Parallel

# The Parallel Construct

## Parallel construct in C/C++

Creates a parallel region by spawning a team of threads



```
#pragma omp parallel [clauses]  
structured-block
```

## Parallel construct in Fortran

```
!$omp parallel [clauses]  
structured-block  
!$omp end parallel
```

# The Parallel Construct

**Parallel construct in C/C++** Optional clause

```
#pragma omp parallel [clauses]  
structured-block
```



**Parallel construct in Fortran**

```
!$omp parallel [clauses]  
structured-block  
!$omp end parallel
```

# The Parallel Construct

## Parallel construct in C/C++

```
#pragma omp parallel [clauses]
```

```
structured-block
```



Block of code

## Parallel construct in Fortran

```
!$omp parallel [clauses]
```

```
structured-block
```

```
!$omp end parallel
```

# OpenMP Runtime Functions

```
#include <omp.h>                                use omp_lib
```

- needed to access declarations of the OpenMP runtime library routines

```
int omp_get_num_threads();                       integer function omp_get_num_threads()
```

- returns the number of threads in the current team
- returns 1 if called outside of a parallel region

```
int omp_get_thread_num();                       integer function omp_get_thread_num()
```

- returns the thread number, within the current team
- returns 0 if called outside of a parallel region



# OpenMP Hello World

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

int main(int argc, char* argv[]) {
    int tid, nthreads;

    nthreads = omp_get_num_threads();

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

        printf("Hello, I'm thread %d of %d.\n",
              tid, nthreads);
    }

    return 0;
}
```

```
program main
    use omp_lib

    integer :: tid, nthreads

    nthreads = omp_get_num_threads()

    !$omp parallel private(tid)
        tid = omp_get_thread_num()

        print 100, tid, nthreads
100    format('Hello, I am thread ', &
&        i0, ' of ', i0)
    !$omp end parallel
end program
```

# Compiling the OpenMP Hello World

To compile an OpenMP program, you need to pass a specific flag to the compiler

<b>GCC</b>	gcc/gfortran	-fopenmp
<b>Clang/AOCC</b>	clang	-fopenmp
<b>Intel classic</b>	icc/ifort	-qopenmp
<b>Intel DPC++</b>	icx	-fopenmp
<b>HPE Cray</b>	cc/ftn	-fopenmp
<b>NVIDIA</b>	nvc/nvfortran	-mp
<b>AMD ROCm</b>	amdclang/amdflang	-fopenmp

This flag instructs the compiler to consider OpenMP directives

# Compiling

For this training we will use the GCC compiler that is available on all CÉCI clusters. Enabling OpenMP with GCC is done by using the `-fopenmp` flag.

These flags may be also be of interest:

---

<code>-march=native</code>	Generate instructions for the compiling machine
<code>-Wall -Wextra</code>	Enable most of the compiler warning
<code>-O3</code>	Enable most optimizations
<code>-Ofast</code>	Enable aggressive optimizations
<code>-fopt-info</code>	Provide an optimization report

---

## Executing the OpenMP Hello World

The basic command to compile the OpenMP hello world is

```
$ gcc -fopenmp -o example omp_helloworld.c
$ OMP_NUM_THREADS=4 ./example
Hello, I'm thread 1 of 4.
Hello, I'm thread 2 of 4.
Hello, I'm thread 3 of 4.
Hello, I'm thread 0 of 4.
```

Notice that we use an environment variable in front of the command to launch our application

## Executing the OpenMP Hello World

The **OMP\_NUM\_THREADS** environment variable sets the number of threads to use for parallel regions

```
export OMP_NUM_THREADS=4 ← 4 threads for the  
./example                duration of the session
```

```
OMP_NUM_THREADS=4 ./example ← 4 threads for this  
                    execution of the program
```

```
OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK ./example ← convenient way to determines  
                                                the number of threads from a  
                                                slurm script
```

## Submitting an OpenMP Job

When submitting your OpenMP job to one of the CÉCI clusters set `cpus-per-task` to specify the number of threads. For example, for NIC5:

```
#!/bin/bash
# Basic submission script for an openmp job
#SBATCH --time=01:00:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem-per-cpu=1024

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

module load releases/2021b
module load GCC/11.2.0

./your_omp_app
```

## Private and Shared Variables

In the Hello World code, we use a `private(tid)` clause to privatize the `tid` variable as each thread need to set its own value.

- `shared(list)` all thread see the same copy of the variables in `list`
- `private(list)` each thread has its own copy of the variables in `list`
- `default(shared|none)` define the default sharing attribute

If you set the default sharing attribute to `none` then, you need to specify the sharing attribute for all the variables used in the parallel region

## Private Variables

- when you privatize a variable it's as if you create a new variable, private variables are uninitialized
- it also means that, at the end of the parallel region the original variable is unchanged
- if you declare a variable inside a parallel region, this variable is private
- in C++ the default constructor is called to create the variable
- if you want to privatize a variable but initialize it with the value before entering the parallel region you can use the `firstprivate(list)` clause
- in C++ the default copy constructor is called to initialize the variable



# Shared Variables

- variables are shared by default
- data allocated on heap (with a `malloc`) are shared and can't be privatized
- shared variables must be handled with care to avoid data races

```
int x = 0;
#pragma omp parallel num_threads(2)
{
    x = x + 1;
}
```

Thread 1	Thread 2		x (in memory)
load x		←	0
x + 1	load x	←	0
store x	x + 1	→	1
	store x	→	1

## Making Things Go Parallel

Creating a parallel region does not mean that that the work will be shared among the threads. For example, if we consider this piece of code:

```
#pragma omp parallel private(tid, niters)
{
  tid = omp_get_thread_num();
  niters = 0;

  for(int i = 0; i < 1000; ++i)
    niters++;

  printf("Number of iterations performed"
        "by thread %d: %d\n", i, niters);
}
```

```
!$omp parallel private(tid, niters)
  tid = omp_get_thread_num()
  niters = 0

  do i = 1,1000
    niters = niters + 1
  end do

  print 100, i, niters
  format('Number of iteration for thread ', &
        i0, ': ', i0)
!$omp end parallel
```

# Making Things Go Parallel

Creating a parallel region does not mean that the work will be shared among the threads. For example, if we consider this piece of code:

```
#pragma omp parallel private(tid, niters)
{
    tid = omp_get_thread_num();
    niters = 0;

    #pragma omp for
    for(int i = 0; i < 1000; ++i)
        niters++;

    printf("Number of iterations performed"
           "by thread %d: %d\n", i, niters);
}
```

```
!$omp parallel private(tid, niters)
    tid = omp_get_thread_num()
    niters = 0

    !$omp for
    do i = 1,1000
        niters = niters + 1
    end do
    !$omp end for

    print 100, i, niters
    100    format('Number of iteration for thread ', &
    &        i0, ': ', i0)
!$omp end parallel
```

## Making Things Go Parallel

```
$ gcc -fopenmp -o example omp_iterations.c  
$ OMP_NUM_THREADS=4 ./example  
Number of iteration for thread 0: 1000  
Number of iteration for thread 1: 1000  
Number of iteration for thread 2: 1000  
Number of iteration for thread 3: 1000
```

**There is no worksharing: all the threads execute all the iterations of the loop**

# Parallel $\neq$ Worksharing

The parallel construct means that

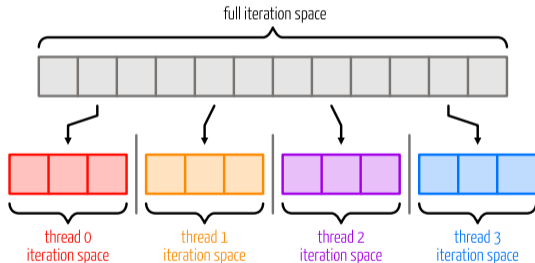
- a team of threads is created, i.e. there is a fork
- the code is executed redundantly by each thread
- the threads in the team join at the end of the region

but

- most scientific workloads can be parallelized by distributing the iterations of a loop among threads
- therefore the parallel construct is not sufficient we need a way to distribute the iterations

# Worksharing

Worksharing a loop is dividing the iteration space into chunks and distribute these chunks to the threads



As the threads run in parallel, we can expect a **nthreads** speedup as each thread works on **niters/nthreads** iterations of the loop

## Distributing iterations

One of the options for sharing the work between the threads is to define lower and higher bounds of the loop depending on the thread ID.

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

    int low = n * tid / nthreads;
    int high = n * (tid + 1) / nthreads;

    for(int i = low; i < high; ++i)
        iterations[tid]++;
}
```

```
!$omp parallel private(tid)
    tid = omp_get_thread_num()
    nthreads = omp_get_num_threads()

    low = n * tid / nthreads + 1
    high = n * (tid + 1) / nthreads

    do i = low, high
        iterations(tid) = iterations(tid) +
            1
    end do
!$omp end parallel
```

## Distributing iterations

```
$ gcc -fopenmp -o example omp_iterations.c
```

```
$ OMP_NUM_THREADS=4 ./example
```

```
Number of iteration for thread 0: 250
```

```
Number of iteration for thread 1: 250
```

```
Number of iteration for thread 2: 250
```

```
Number of iteration for thread 3: 250
```



## Distributing iterations with a directive

Instead of computing the bounds, we can use the **for** (or **do**) construct.

```
for(int i = 0; i < max_threads; ++i)
    iterations[i] = 0;
```

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < n; ++i)
        iterations[tid]++;
}
```

```
!$omp parallel private(tid)
!$omp do
do i = 1, n
    iterations(tid) = iterations(tid) +
        1
end do
!$omp end do
!$omp end parallel
```

## Distributing iterations with a directive

```
$ gcc -fopenmp -o example omp_for_iters.c
```

```
$ OMP_NUM_THREADS=4 ./example
```

```
Number of iteration for thread 0: 250
```

```
Number of iteration for thread 1: 250
```

```
Number of iteration for thread 2: 250
```

```
Number of iteration for thread 3: 250
```

## The Canonical for-loop

The for-loop needs to be in canonical form to be used with the **for** directive

```
#pragma omp for
for ([inttype] var = start; var < end; ++var
    <=      var++
    >      var += incr
    >=     var = var + incr
    var--, ...)
```

- **var**, cannot be modified in the loop body. It must be an integer (signed or unsigned), a pointer or random access iterator type
- **start**, **end** and **incr** must be loop invariant expressions, the number of iterations must be computable when the loop begins

## Parallel Region Binding

In order for the iterations of a loop to be shared among the threads by a `for/do`, the construct needs a parallel region to bind to. If we take the previous example and remove the parallel region:

```
int max_threads = omp_get_max_threads();
int* iterations = malloc(sizeof(int)*max_threads);

for(int i = 0; i < max_threads; ++i)
    iterations[i] = 0;

#pragma omp for
for(int i = 0; i < n; ++i)
    iterations[tid]++;

for(int i = 0; i < max_threads; ++i)
    printf("Number of iteration for thread %d: %d\n",
        i, iterations[i]);
```

```
max_threads = omp_get_max_threads()
allocate(iterations(0:max_threads-1))

iterations = 0

!$omp do
    do i = 1, n
        iterations(tid) = iterations(tid) + 1
    end do
!$omp end do

do i = 0, max_threads-1
    print 100, i, iterations(i)
    format('Number of iteration for thread ', i0, &
        ': ', i0)
end do
```

## Parallel Region Binding

```
$ OMP_NUM_THREADS=4 ./example  
Number of iteration for thread 0: 1000  
Number of iteration for thread 1: 0  
Number of iteration for thread 2: 0  
Number of iteration for thread 3: 0
```

As there was no parallel region to bind to, the **for/do** construct binds to the master thread.

## Combined Directive

The following code snippet,

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

```
!$omp parallel
  !$omp do
    do i = 1,n
      call do_something()
    end do
  !$omp end do
!$omp end parallel
```

may also be written as combined **parallel** and **for** directives

```
#pragma omp parallel for
for(int i = 0; i < n; ++i)
  do_something();
```

```
!$omp parallel do
  do i = 1,n
    call do_something()
  end do
!$omp end parallel do
```

## Orphaning

- Directives are active in the dynamic scope of a parallel region, not just its lexical scope. This allows for orphaned directives.
- Orphaning is a situation when directives related to a parallel region are outside the lexical extent of the parallel region.
- Typical situation is calling a function containing a worksharing directive from a parallel region.

# Orphaning Example

```
void ax(int n, double alpha, double* x) {
    int nthreads = omp_get_num_threads();
    int tid = omp_get_thread_num();

    printf("Executing ax by thread %d of %d threads.\n", tid,
           nthreads);

    int niters = 0;

    #pragma omp for
    for (int i = 0; i < n; ++i) {
        x[i] = alpha * x[i];
        niters++;
    }

    printf("Thread with id %d did %d iterations.\n", tid, niters);
}

int main ( int argc, char *argv[] ) {
    // [...]

    #pragma omp parallel
    {
        ax(n, 3.0, x);
    }

    ax(n, 5.0, y);
}
```

```
!$omp parallel
    call ax(n, 3.0d0, x)
!$omp end parallel

call ax(n, 5.0d0, y)

! [...]

contains
    subroutine ax(n, alpha, x)
        ! [...]

        print 100, tid, nthreads
        format('Executing ax by thread ', i0,
              ' of ', i0, ' threads.')

!$omp do
    do i = 1,n
        x(i) = alpha * x(i)
        niters = niters + 1
    end do
!$omp end do

    print 200, tid, niters
    format('Thread with id ', i0, &
          ' did ', i0, ' iterations.')

```



## Orphaning Example

```
$ gcc -fopenmp -o example omp_orphaned.c
$ OMP_NUM_THREADS=4 ./example
Executing ax by thread 0 of 4 threads.
Executing ax by thread 2 of 4 threads.
Executing ax by thread 1 of 4 threads.
Executing ax by thread 3 of 4 threads.
Thread with id 0 did 250 iterations.
Thread with id 1 did 250 iterations.
Thread with id 2 did 250 iterations.
Thread with id 3 did 250 iterations.
Executing ax by thread 0 of 1 threads.
Thread with id 0 did 1000 iterations.
```

## Loop collapsing

In some cases, you can collapse the loops into one in order to increase the run trip of the loop.

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < n; ++j) {
        a[i][j] = do_something();
    }
}
```

```
!$omp parallel do collapse(2)
    do j = 1,3
        do i = 1,n
            a(i, j) = do_something()
        end do
    end do
!$omp end parallel do
```

This is particularly useful when one of the loops is not of sufficient length to have efficient parallelization.

## Loop collapsing

The **collapse** clause, collapse the iterations of the n-associated loops to which the clause applies into one larger iteration space. This clause can only apply on tightly nested loops, meaning that there is no code between the loops.

```
#pragma omp for collapse(n)  
    nested-for-loops
```

```
!$omp do collapse(n)  
    nested-do-loops
```

## Loop Scheduling

Loop scheduling, specify how iterations of a loop are divided into contiguous non-empty subsets (chunks), and how these chunks are distributed to the threads. Changing the loop scheduling is possible to use the **schedule** clause.

```
#pragma omp for schedule(kind, chunk)
  for-loop
```

```
!$omp do schedule(kind, chunk)
  do-loop
!$omp end do
```

Where the value of **kind** can be **static**, **dynamic**, **guided** or **runtime**. The default scheduling is **static**. The optional **chunk** may have different behaviour depending on the scheduling.

## Static Loop Scheduling

With **static** loop scheduling, iterations are divided into chunks and the chunks are assigned to the threads. Each chunk contains the same number of iterations, except for the chunk that contains the last iteration, which may have fewer iterations.

```
#pragma omp for schedule(static)
  for-loop
```

```
!$omp do schedule(static)
  do-loop
!$omp end do
```

You can also provide a chunk size

```
#pragma omp for schedule(static, 100)
  for-loop
```

```
!$omp do schedule(static, 100)
  do-loop
!$omp end do
```

## Dynamic Loop Scheduling

With **dynamic** loop scheduling, the iterations are distributed to threads in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.

```
#pragma omp for schedule(dynamic)
  for-loop
```

```
!$omp do schedule(dynamic)
  do-loop
!$omp end do
```

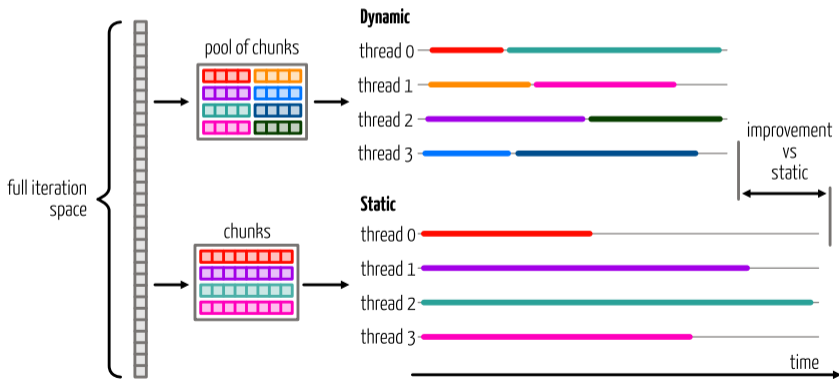
You can also provide a chunk size

```
#pragma omp for schedule(dynamic, 100)
  for-loop
```

```
!$omp do schedule(dynamic, 100)
  do-loop
!$omp end do
```

# Dynamic Loop Scheduling

Dynamic scheduling particularly relevant when the amount of work of the loop iteration is not constant.

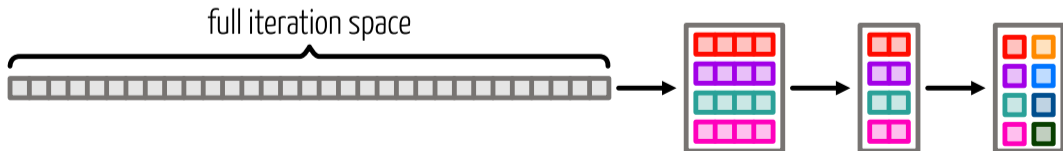


## Guided Loop Scheduling

The **guided** loop scheduling is similar to the **dynamic** scheduling except that the size of each chunk is proportional to the number of unassigned iterations, decreasing to one.

```
#pragma omp for schedule(guided)  
for-loop
```

```
!$omp do schedule(guided)  
do-loop  
!$omp end do
```





## Why Using the Scheduling Clause?

- The default scheduling, `static` with a `chunk` size equals to `niter/nthreads` is not ideal for all workload.
- It may be the case that iterations of high index represent more work. In that case, some of the threads will finish early and have nothing to do. We have a load imbalance.
- Changing the scheduling may help balance the amount of work between the threads.

## Example: Number of Prime Numbers

```
int prime, sum = 0;
#pragma omp parallel shared(n) private(prime)
{
    #pragma omp for reduction(+:sum)
    for (int i = 2; i <= n; i++) {
        prime = 1;

        for (int j = 2; j < i; j++) {
            if ( i % j == 0 ) {
                prime = 0;
                break;
            }
        }

        sum += prime;
    }
}
```

Trip count of this loop may be very low or  
very high depending if the number is prime  
or not

## Example: Number of Prime Numbers

```
int prime, sum = 0;
#pragma omp parallel shared(n) private(prime)
{
    #pragma omp for reduction(+:sum)
    for (int i = 2; i <= n; i++) {
        prime = 1;

        for (int j = 2; j < i; j++) {
            if ( i % j == 0 ) {
                prime = 0;
                break; ← If the number is not a prime number, we have an early exit
            }
        }

        sum += prime;
    }
}
```

## Example: Number of Prime Numbers

```
$ gcc -fopenmp -o example omp_schedule_prime.c
$ OMP_NUM_THREADS=4 ./example
```

N	Pi(N)	Default Time	Static Time	Dynamic Time	Guided Time
1024	172	0.000182	0.000120	0.000104	0.000121
2048	309	0.000561	0.000359	0.000425	0.000393
4096	564	0.001987	0.001309	0.001216	0.001239
8192	1028	0.007116	0.004474	0.004375	0.005114
16384	1900	0.029730	0.015594	0.015902	0.015161
32768	3512	0.099248	0.058475	0.056940	0.057160
65536	6542	0.358250	0.218291	0.244626	0.254815
131072	12251	1.416871	0.848736	0.788619	0.819390
262144	23000	5.207946	3.193940	3.062080	3.064527
524288	43390	20.565462	12.638959	12.086839	12.102800

## Example: Triangular Loop

```
#pragma omp parallel shared(a, n)
{
    #pragma omp for
    for (int i = 0; i < n; ++i) {
        a[i] = 0.0;

        for (int j = 0; j < i; ++j) {
            a[i] += cos( -3.1 * sin( 2.3 * cos ( (double) j ))) ;
        }
    }
}
```

## Example: Triangular Loop

```
$ gcc -fopenmp -o example omp_schedule_triangular.c  
$ OMP_NUM_THREADS=4 ./example
```

N	Default Time	Static Time	Dynamic Time	Guided Time
1024	0.025865	0.016811	0.018739	0.018241
2048	0.100062	0.070023	0.082587	0.091206
4096	0.383107	0.238520	0.232556	0.226914
8192	1.515341	0.905186	0.895541	0.880046
16384	6.064787	3.540685	3.526388	3.590453
32768	24.041088	15.465762	14.088137	14.539937
65536	97.495829	59.291353	59.403173	60.252156

# Synchronization

# Synchronization

Synchronization ensures that two or more threads do not simultaneously execute some part of the program.

Synchronization may be needed for various reasons:

- makes sure that a particular operation is only executed once
- to avoid conflicts when accessing shared data
- ensure the order in which tasks are executed



# Barrier

A **barrier** directive is a synchronization point at which the threads in a parallel region will wait until all other threads in that section reach the same point.

- When a first thread reaches the barrier, it waits
- When a second thread reaches the barrier, it does the same thing and so on
- When the last thread reaches the barrier, all the threads resume execution

## Barrier

Most common usage of a barrier is to make sure that the value set by a thread is correctly defined before reading it from another thread.

```
#pragma omp parallel private(tid, neighb)
{
    tid = omp_get_thread_num();
    neighb = tid - 1;

    if (tid == 0)
        neighb = omp_get_num_threads() - 1;

    a[tid] = a[tid] * 3.5;

    #pragma omp barrier

    b[tid] = a[neighb] + c;
}
```

```
!$omp parallel private(tid, neighb, nthreads)
    tid = omp_get_thread_num()
    nthreads = omp_get_num_threads()
    neighb = tid - 1

    if (tid .eq. 0) then
        neighb = nthreads - 1
    end if

    a(tid) = a(tid) * 3.5

    !$omp barrier

    b(tid) = a(neighb) + c
!$omp end parallel
```

## Implicit Barrier

Some constructs in OpenMP have an implicit barrier. This is the case for the `parallel` and `for/do` constructs.

```
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < n; ++i) {

    // ...

  } ← Implicit barrier, wait for all the threads to finish their iterations

  // ...

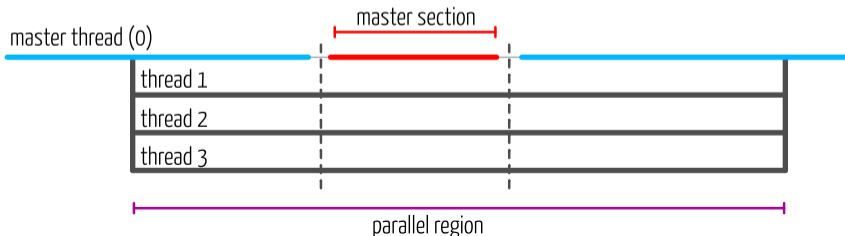
} ← Implicit barrier, wait for all the threads to join
```

## Master Directive

A **master** construct specifies a block of code that should be executed only by the master thread of the team.

```
#pragma omp master  
structured-block
```

```
!$omp master  
structured-block  
!$omp end master
```



# Hello World, Master Edition

Let's revisit the hello world program but, this time, only the master thread print the number of threads in the team.

```
#pragma omp parallel
{
    printf("Hello, I'm thread %d\n",
           omp_get_thread_num());

    #pragma omp master
    {
        printf("There is %d threads in the team\n",
               omp_get_num_threads());
    }
}
```

```
!$omp parallel
    print 100, omp_get_thread_num()
100    format('Hello, I am thread ', i0)

    !$omp master
        print 200, omp_get_num_threads()
200    format('There is ', i0, &
&        ' threads in the team')
    !$omp end master
!$omp end parallel
```

## Hello World, Master Edition

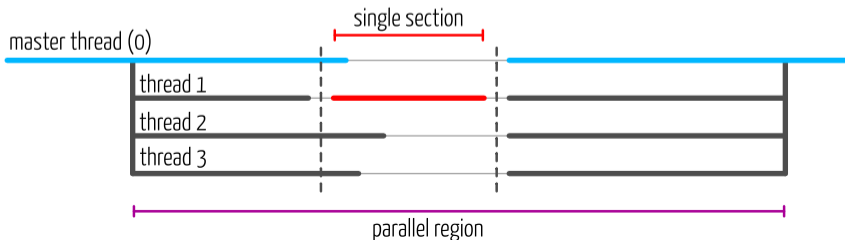
```
$ gcc -fopenmp -o example omp_helloworld_master.c
$ OMP_NUM_THREADS=4 ./example
Hello, I'm thread 3
Hello, I'm thread 0
There is 4 threads in the team
Hello, I'm thread 2
Hello, I'm thread 1
```

## Single Directive

A **single** directive is executed by only one of the threads in the team (not necessarily the master thread). There is an implicit barrier at the end.

```
#pragma omp single  
  structured-block
```

```
!$omp single  
  structured-block  
!$omp end single
```



## Hello World, Single Edition

Let's revisit the hello world program using the single construct. This time we illustrate the most common usage of the single construct, that is, assign a value to a shared variable.

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

    #pragma omp single
    {
        nthreads = omp_get_num_threads();

        printf("Hello, I'm thread %d of %d"
              " in the single construct.\n",
              tid, nthreads);
    }

    printf("Hello, I'm thread %d of %d.\n",
          tid, nthreads);
}
```

```
!$omp parallel private(tid)
    tid = omp_get_thread_num()

    !$omp single
        nthreads = omp_get_num_threads()

        print 100, tid, nthreads
        format('Hello, I am thread ', i0, &
              ' of ', i0, &
              ' in the single construct.')
    !$omp end single

    print 200, tid, nthreads
    format('Hello, I am thread ', i0, &
          ' of ', i0, '.')
!$omp end parallel
```



## Hello World, Single Edition

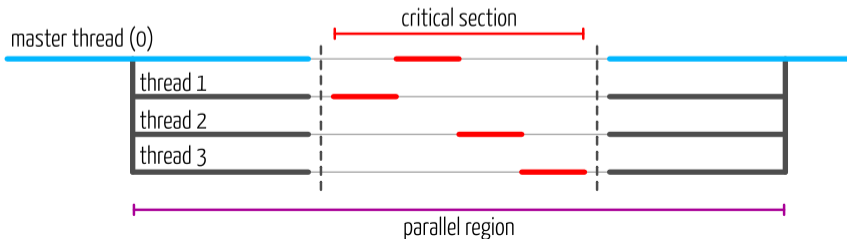
```
$ gcc -fopenmp -o example omp_helloworld_single.c
$ OMP_NUM_THREADS=4 ./example
Hello, I'm thread 3 of 4 in the single construct.
Hello, I'm thread 3 of 4.
Hello, I'm thread 2 of 4.
Hello, I'm thread 0 of 4.
Hello, I'm thread 1 of 4.
```

# Critical Section

A **critical** section restricts execution of the associated structured block to one thread at a time.

```
#pragma omp critical  
structured-block
```

```
!$omp critical  
structured-block  
!$omp end critical
```



# Critical Section

Critical section is mostly used to update shared variables avoiding a data race.

```
#pragma omp parallel private(tid, local_sum)
{
    tid = omp_get_thread_num();
    local_sum = 0;

    #pragma omp for
    for (int i = 0; i < n; ++i)
        local_sum += a[i];

    #pragma omp critical
    {
        sum += local_sum;
        printf("Thread %d: local sum = %d, sum = %d.\n",
            tid, local_sum, sum);
    }
}

printf("Sum after parallel region: %d.\n", sum);
```

```
!$omp parallel private(tid, local_sum)
    tid = omp_get_thread_num()
    local_sum = 0

    !$omp do
        do i = 1,n
            local_sum = local_sum + a(i)
        end do
    !$omp end do

    !$omp critical
        global_sum = global_sum + local_sum

        print 100, tid, local_sum, global_sum
        format('Thread ', i0, ': local sum = ', i0, &
            ', sum = ', i0, '.')
    !$omp end critical
!$omp end parallel

print*, 'Sum after parallel region:', global_sum
```

## Critical Section

Critical section is mostly used to update shared variables avoiding a data race.

```
#pragma omp parallel shared(sum) private(tid, local_sum)
{
    tid = omp_get_thread_num();
    local_sum = 0;

    #pragma omp for
    for (int i = 0; i < n; ++i)
        local_sum += a[i];

    #pragma omp critical
    {
        sum += local_sum;
        printf("Thread %d: local sum = %d, sum = %d.\n",
            tid, local_sum, sum);
    }
}
```

← Critical section to update the global sum. Without the critical section, there is a potential data race here

```
printf("Sum after parallel region: %d.\n", sum);
```

## Critical Section

```
$ gcc -fopenmp -o example omp_critical.c
$ OMP_NUM_THREADS=4 ./example
Thread 0: local sum = 300, sum = 300.
Thread 3: local sum = 2175, sum = 2475.
Thread 1: local sum = 925, sum = 3400.
Thread 2: local sum = 1550, sum = 4950.
Sum after parallel region: 4950.
```

## Named Critical Section

Optional name clause



```
#pragma omp critical (name)  
structured-block
```

- A thread waits at the beginning of a critical section until no other thread is executing a critical section with the same name
- All unnamed critical directives map to the same name
- Critical section names are global to the program

## Named Critical Section

```
#pragma omp critical (sum)
{
    sum += local_sum;
    printf("Thread %d: local sum = %d,"
           " sum = %d.\n",
           tid, local_sum, sum);
}
```

```
#pragma omp critical (max)
{
    max = MAX(max, local_max);
    printf("Thread %d: local max = %d,"
           " max = %d.\n",
           tid, local_max, max);
}
```

```
!$omp critical (sum)
    global_sum = global_sum + local_sum
    ;
    print 100, tid, 'sum', local_sum, &
    &                                'sum', global_sum
!$omp end critical (sum)
```

```
!$omp critical (max)
    global_max = max(global_max,
                     local_max)
    print 100, tid, 'max', local_max, &
    &                                'max', global_max
!$omp end critical (max)
```

## Named Critical Section

```
#pragma omp critical (sum) ← First critical section for the global sum
{
    sum += local_sum;
    printf("Thread %d: local sum = %d, sum = %d.\n",
        tid, local_sum, sum);
}
```

```
#pragma omp critical (max)
{
    max = MAX(max, local_max);
    printf("Thread %d: local max = %d, max = %d.\n",
        tid, local_max, max);
}
```



## Named Critical Section

```
#pragma omp critical (sum) ← First critical section for the global sum
{
    sum += local_sum;
    printf("Thread %d: local sum = %d, sum = %d.\n",
        tid, local_sum, sum);
}
```

```
#pragma omp critical (max) ← Second critical section for the global maximum.
                               A thread can be in the first section while an other
                               is in the second one
{
    max = MAX(max, local_max);
    printf("Thread %d: local max = %d, max = %d.\n",
        tid, local_max, max);
}
```

## Named Critical Section

```
$ gcc -fopenmp -o example omp_critical_named.c
$ OMP_NUM_THREADS=4 ./example
Thread 3: local sum = 2175, sum = 2175.
Thread 3: local max = 99, max = 99.
Thread 1: local sum = 925, sum = 3100.
Thread 1: local max = 49, max = 99.
Thread 2: local sum = 1550, sum = 4650.
Thread 0: local sum = 300, sum = 4950.
Thread 2: local max = 74, max = 99.
Thread 0: local max = 24, max = 99.
Sum after parallel region: 4950.
Max after parallel region: 99.
```

## The `nowait` Clause

```
#pragma omp for
for (int i = 0; i < n; ++i) {
    local_sum += a[i];
}
```

← There is an implicit barrier here

```
#pragma omp critical
{
    sum += local_sum;
    printf("Thread %d: local sum = %d, sum = %d.\n",
        tid, local_sum, sum);
}
```

## The `nowait` Clause

```
#pragma omp for
for (int i = 0; i < n; ++i) {
    local_sum += a[i];
}
```

```
#pragma omp critical
```

← There is no need to wait for the other threads to finish  
the iterations to execute the critical section

```
{
    sum += local_sum;
    printf("Thread %d: local sum = %d, sum = %d.\n",
        tid, local_sum, sum);
}
```

## The `nowait` Clause

```
#pragma omp for nowait ← We add a nowait clause to the directive  
for (int i = 0; i < n; ++i) {  
    local_sum += a[i];  
} ← The implicit barrier at the end of the loop is lifted
```

```
#pragma omp critical  
{  
    sum += local_sum;  
    printf("Thread %d: local sum = %d, sum = %d.\n",  
        tid, local_sum, sum);  
}
```

## The `nowait` Clause

The `nowait` clause applied to a `for` construct remove the implicit barrier at the end of the construct.

```
#pragma omp for nowait  
    structured-block
```

```
!$omp do  
    structured-block  
!$omp end do nowait
```

The `nowait` clause can also be applied to a `single` directive.

```
#pragma omp single nowait  
    structured-block
```

```
!$omp do  
    structured-block  
!$omp end single nowait
```

## The `nowait` Clause

The `nowait` clause can also be convenient when the work in two different loops are independent from each other.

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

    #pragma omp for nowait
    for (int i = 0; i < n; ++i) {
        e[i] = a[i] + c[i];
    }
}
```

```
!$omp parallel
    !$omp do
        do i = 1,n
            d(i) = a(i) + b(i)
        end do
    !$omp end do nowait

    !$omp do
        do i = 1,n
            e(i) = a(i) + c(i)
        end do
    !$omp end do nowait
!$omp end parallel
```

# The `nowait` Clause

The `nowait` clause can also be convenient when the work in two different loops are independent from each other.

```
#pragma omp parallel
```

```
{
```

```
    #pragma omp for nowait
```

```
    for (int i = 0; i < n; ++i) {
```

```
        d[i] = a[i] + b[i];
```

```
    } ← No barrier at the end of the loop
```

```
    #pragma omp for nowait
```

```
    for (int i = 0; i < n; ++i) {
```

```
        e[i] = a[i] + c[i];
```

```
    }
```

```
}
```

← The threads start the iterations of this loop as soon as they finish their work in the first loop



## The `nowait` Clause

The `nowait` clause can also be convenient when the work in two different loops are independent from each other.

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

    #pragma omp for nowait
    for (int i = 0; i < n; ++i) {
        e[i] = a[i] + c[i];
    }
} ← Implicit barrier at the end of the parallel region
```

## Reduction

The `reduction` clause avoid data races when summing or combining values. This clause can be applied to the `parallel` and `for` constructs

```
reduction(op:list)
```

`op` is an operator:

- Arithmetic reductions: `+` `*` `-` `max` `min`
- Logical operator reductions: `&` `&&` `|` `||`

# Reduction

The sum and maximum example using critical region can be rewritten with **reduction** clauses instead

```
#pragma omp parallel for reduction(+:sum) \  
                        reduction(max:max)  
for (int i = 0; i < n; ++i) {  
    sum += a[i];  
    max = MAX(max, a[i]);  
}  
  
printf("Sum after parallel region: %d.\n", sum);  
printf("Max after parallel region: %d.\n", max);
```

```
!$omp parallel for reduction(+:sum) &  
!$omp&                        reduction(max:max)  
    do i = 1,n  
        sum += sum + a(i)  
        imax = max(imax, a(i))  
    end do  
!$omp end parallel for  
  
print*, 'Sum after parallel region: ', sum  
print*, 'Max after parallel region: ', imax
```

## Atomic operation

An atomic operation is an operation that will always be executed without any other thread being able to read or change state that is read or changed during the operation.

```
#pragma omp atomic [atomic-clause]  
expression-statement
```

## Atomic operation

```
#pragma omp atomic atomic-clause  
expression-statement
```

The value of **atomic-clause** can be one of the following: **read**, **write**, **update** and **capture**. If no **atomic-clause** is specified, the default value is **update**.

## Atomic operation: Read and Write

The **read** clause allows for the atomic read of **x**.

```
#pragma omp atomic read  
v = x;
```

The **write** clause allows for the atomic write of **x**. Here, **expr** is an expression with scalar type, i.e. the result of the expression is a scalar.

```
#pragma omp atomic write  
x = expr;
```

## Atomic operation: Update

The **update** clause allows for the atomic update of **x**.

```
#pragma omp atomic update  
expression-statement
```

Expression statement

---

`x++;`

`x--;`

`++x;`

`--x;`

`x op= expr;` `x = x op expr;` `x = expr op x;`

---

## Atomic operation: Capture

The **capture** clause allows for atomic update of the location designated by **x** while also capturing the original or final value of the location designated by **x**.

```
#pragma omp atomic capture  
expression-statement
```

Expression statement

---

```
v = x++; v = x--; v = ++x; v = --x;
```

```
v = x op= expr; v = x = x op expr;
```

```
v = x = expr op x;
```

---



## Atomic operation: Capture

The **capture** clause allows for atomic update of the location designated by **x** while also capturing the original or final value of the location designated by **x**.

```
#pragma omp atomic update  
structured-block
```

### Structured block (part. 1)

---

```
{ v = x; x op= expr; }      { x op= expr; v = x; }
```

```
{ v = x; x = x op expr; }  { v = x; x = expr op x; }
```

```
{ x = x op expr; v = x; }  { x = expr op x; v = x; }
```

---

## Atomic operation: Capture

The **capture** clause allows for atomic update of the location designated by **x** while also capturing the original or final value of the location designated by **x**.

```
#pragma omp atomic update  
structured-block
```

### Structured block (part. 2)

---

```
{ v = x; x++; } { v = x; ++x; } { ++x; v = x; }
```

```
{ x++; v = x; } { v = x; x--; } { v = x; --x; }
```

```
{ --x; v = x; } { x--; v = x; }
```

---

## Atomic example

The previous example of the summation of the elements of an array using a **critical** construct can be rewritten using an **atomic** update.

```
#pragma omp for
for (int i = 0; i < n; ++i) {
    local_sum += a[i];
}
```

```
#pragma omp atomic
sum += local_sum;
```

```
!$omp do
do i = 1,n
    local_sum += local_sum + a(i)
end do
!$omp end do

!$omp atomic
sum = sum + local_sum
!$omp end atomic
```

## Atomic example

```
for (i = 0; i < 10000; ++i) {  
    index[i] = i % 1000;  
}
```

```
for (i = 0; i < 1000; ++i)  
    x[i] = 0.0;
```

```
#pragma omp parallel for  
for (i = 0; i < 10000; ++i) {  
    #pragma omp atomic update  
    x[index[i]] += 1.0 * i;  
}
```

```
do i = 1,10000  
    inds(i) = mod(i, 1000)  
end do
```

```
do i = 1,1000  
    x(i) = 0.0  
end do
```

```
!$omp parallel do  
    do i = 1,10000  
        !$omp atomic update  
        x(inds(i)) = x(inds(i)) + 1.0 * i  
    end do  
!$omp end parallel do
```

## Atomic example

```
for (i = 0; i < 10000; ++i) {  
    index[i] = i % 1000;  
}
```

```
for (i = 0; i < 1000; ++i)  
    x[i] = 0.0;
```

```
#pragma omp parallel for  
for (i = 0; i < 10000; ++i) {  
    #pragma omp atomic update  
    x[index[i]] += 1.0 * i;  
}
```

The advantage of using **atomic** in this example is that it allows updates of two different elements of **x** in parallel. If a **critical** construct were used, all updates to elements of **x** would be executed serially

## Atomic vs. Critical

Safely increasing the value of `count` in parallel can be done either by using an `atomic` or a `critical` directive

```
#pragma omp atomic  
count++;
```

- An atomic operation has much lower overhead but the set of possible operations is restricted
- It can take advantage of hardware support for atomic operations

```
#pragma omp critical  
count++;
```

- A critical section can surround any arbitrary block of code
- There is a significant overhead when a thread enters and exits the critical section

# Atomic vs. Reduction

Don't use **atomic** operation this way:

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    #pragma omp atomic
    sum += a[i];
}
```

It is better to use a **reduction** clause:

```
#pragma omp parallel for reduction(+sum)
for (int i = 0; i < n; ++i) {
    sum += a[i];
}
```

```
!$omp parallel do
do i = 1,n
    !$omp atomic
    sum = sum + a(i)
end do
!$omp end parallel do
```

```
!$omp parallel do reduction(+sum)
do i = 1,n
    sum = sum + a(i)
end do
!$omp end parallel do
```

## Performance Considerations

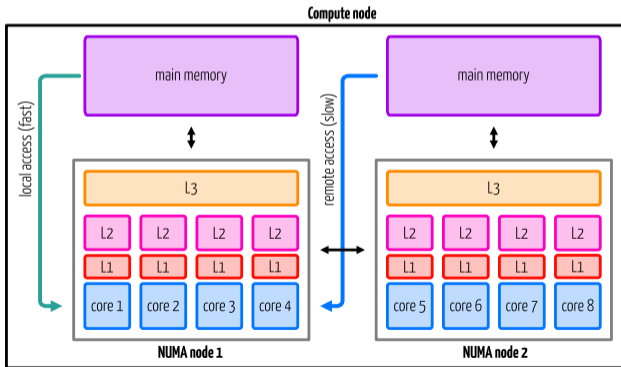
- Avoid or minimize the use of `barrier` and `critical` sections.
- Use the `nowait` clause where possible to eliminate unnecessary barriers
- Favour the use of `master` instead of `single`



# OpenMP and NUMA

# NUMA

Non-uniform memory access (NUMA) is a memory design where the memory access time depends on the memory location relative to the NUMA node



- access to the memory within the same NUMA node is faster (local access)
- access to the memory outside of the NUMA node is slower (remote access)

## OpenMP and cc-NUMA

```
double* A = (double*)malloc(N * sizeof(double));
```

```
#pragma omp parallel for  
for (int i = 0; i < N; i++) {  
    A[i] = 0.0;  
}
```

- For a serial code: all array elements are allocated in the memory of the NUMA node containing the core executing the thread
- For a parallel code on an OS with a first touch policy the array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing

## OpenMP and cc-NUMA

You also have two options for the placement of your threads. The first is put the threads far apart, i.e. on different sockets.

- may improve the aggregated memory bandwidth available to your application
- may improve the combined cache size available to your application
- may decrease performance of synchronization constructs

The second option is to put the threads close together, i.e. on two adjacent cores.

- may improve performance of synchronization constructs
- may decrease the available memory bandwidth and cache size

## OpenMP and cc-NUMA

For the placement, you can use the `OMP_PROC_BIND` environment variable with the values:

- `close`: successively through the available places
- `spread`: which spreads the threads over the places

The second option is the `OMP_PLACES` environment variable with the values:

- `core`: places correspond to the cores
- `socket`: places correspond to the sockets

# False Sharing

## False Sharing in Action

Another thing you need to consider if you want to get the best out of OpenMP is false sharing. To discuss this we will start with this piece of code:

```
double local_sum[omp_get_max_threads()];
double sum = 0.0;

#pragma omp parallel shared(sum)
{
    int tid = omp_get_thread_num();
    local_sum[tid] = 0.0;

    #pragma omp omp for
    for (int i = 0; i < n; ++i)
        local_sum[tid] += 0.5 * x[i] + y[i];

    #pragma omp atomic
    sum += local_sum[tid];
}
```

## False Sharing in Action

Let's measure the time spend in the parallel region (using the `omp_get_wtime()` function).

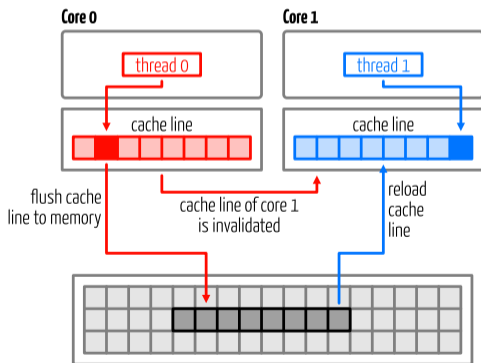
Threads	Time (s)
1	0.535418
2	0.421140
4	0.554419
8	0.597622

- The speedup from 1 thread to 2 threads is bad
- When going to 4 and 8 threads the time spend in the parallel region is worse than with 1 thread



## False Sharing

False sharing is when threads impact the performance of each other while modifying independent variables sharing the same cache line



- If one core writes, the cache line holding the memory line is invalidated on other cores.
- Even though another core is not using that data, the second core will need to reload the line before it can access its own data again.

## False Sharing: Solution

Solution: introduce a padding.

```
double local_sum[LINESIZE*omp_get_max_threads()];
double sum = 0.0;

#pragma omp parallel shared(sum)
{
    int tid = omp_get_thread_num();
    local_sum[LINESIZE*tid] = 0.0;

    #pragma omp omp for
    for (int i = 0; i < n; ++i)
        local_sum[LINESIZE*tid] += 0.5 * x[i] + y[i];

    #pragma omp atomic
    sum += local_sum[LINESIZE*tid];
}
```

## False Sharing: Solution

Timing for different padding on a CPU with a cache line size of 64 bytes.

Threads	Time (s)	Time (s)	Time (s)
	padding = 4	padding = 8	padding = 16
1	0.535418	0.535418	0.535418
2	0.601417	0.270089	0.270843
4	0.441149	0.152651	0.149363

## False sharing

- When threads access global or dynamically allocated shared data structures there is a potential source of false sharing
- False sharing may be difficult to spot. For example, when threads access completely different global variables that happen to be relatively close together in memory.
- Use thread-local copies of data when possible. The thread-local copy can be read and frequently modified and only when complete, be copied back to the global data structure