OpenMP for High-performance computing

N. Shukla

High-Performance Computing Department CINECA

Casalecchio di Reno Bologna, Italy

Email: n.shukla@cineca.it

Preliminaries- I

Acknowledgements

- **Tim Mattson** (Intel)
- **Tom Deakin** (University of Bristol)
- **Michael Klemm** (Intel)
- and many others
- OpenMP 5.0.1 specification and examples https://www.openmp.org/resources/

Download code and slides

https://gitlab.hpc.cineca.it/progpu/gpuprogramming.git

Preliminaries-2: disclosure

Goal is to learn OpenMP 4.5

from basic to advanced topics

Methodology:

mixtures of lectures and exercises (total 5)

Materials:

hands on tutorial, exercises and solution

Resources:

Galileo / M100 (get your account if you still do not have)

Target audience

Level of the content is aimed for beginners/ intermediate students

Prior knowledge:

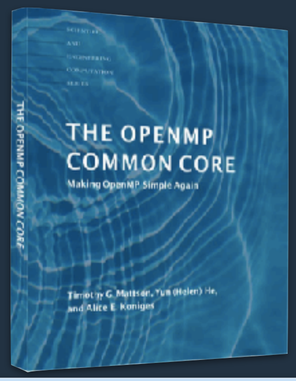
Fortran, C/C++ (basic level)

MPI parallel programming with MPI useful

OpenMP was born to satisfy the need of unification of proprietary solutions

- October 97 : OpenMP 1.0 for Fortran
- October 98 : OpenMP 1.0 for C /C++
- October 99 : OpenMP 1.0 for Fortran
- November 00 : OpenMP 2.0 for Fortran
- November 02 : OpenMP 2.0 for C /C++
- May 05 : OpenMP 2.0 for C/C++ & Fortran
- May 08 : OpenMP 3.0 for C/C++ & Fortran
- July 11 : OpenMP 3.1 for C/C++ & Fortran
- July 13 : OpenMP 4.0
- November 15 : OpenMP 4.5
- November 18 : OpenMP 5.0
- November 20 : OpenMP 5.1

Reference books About OpenMP



A new book about the OpenMP Common Core, 2019

A printed copy of the 5.0 specifications, 2019

A book that covers all of the OpenMP 4.5 features, 2017

Topics we will cover?

- Before starting parallel paradigms
- First steps with OpenMP
- Parallel construct
- Worksharing constructs
- Data sharing
- Synchronization Constructs
- OpenMP for SIMD vectorization

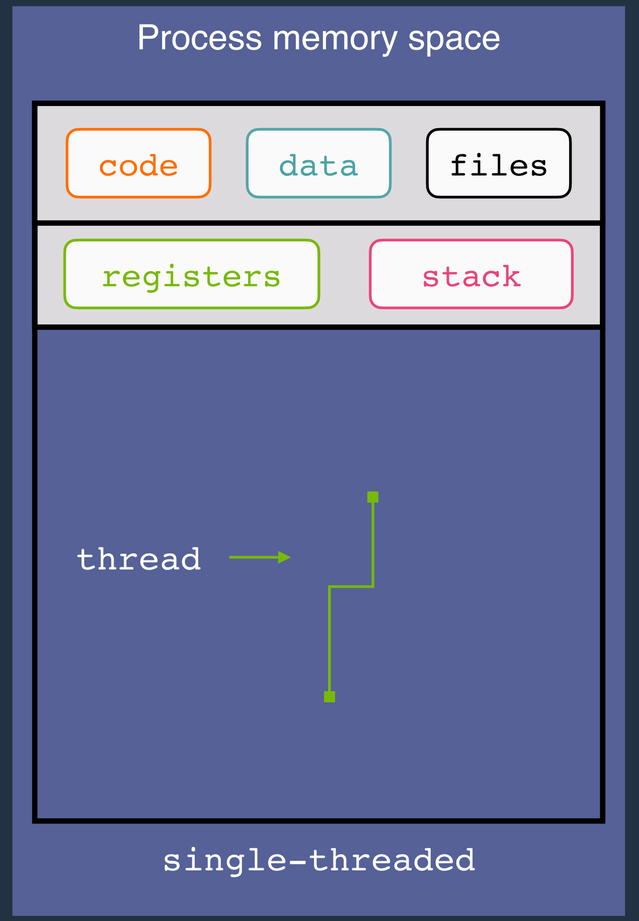
Let’s clear off some terminology: Process vs Threads

Process is an active program i.e a program that is under execution

- Text: Machine code
- Data: Global variables
- Stack: Local variables
- Program counter (PC): A pointer to the instructions to be executed
- Heap: All dynamically allocated data

The process contains several concurrent execution flows (threads)

- A process starts with a single thread (primary thread)
- Later, it can create more than one thread from any of its threads
- All threads of a process have access to its memory and system resource



What is a thread?

Thread is the smallest unit of execution to which processor allocates time and consists of:

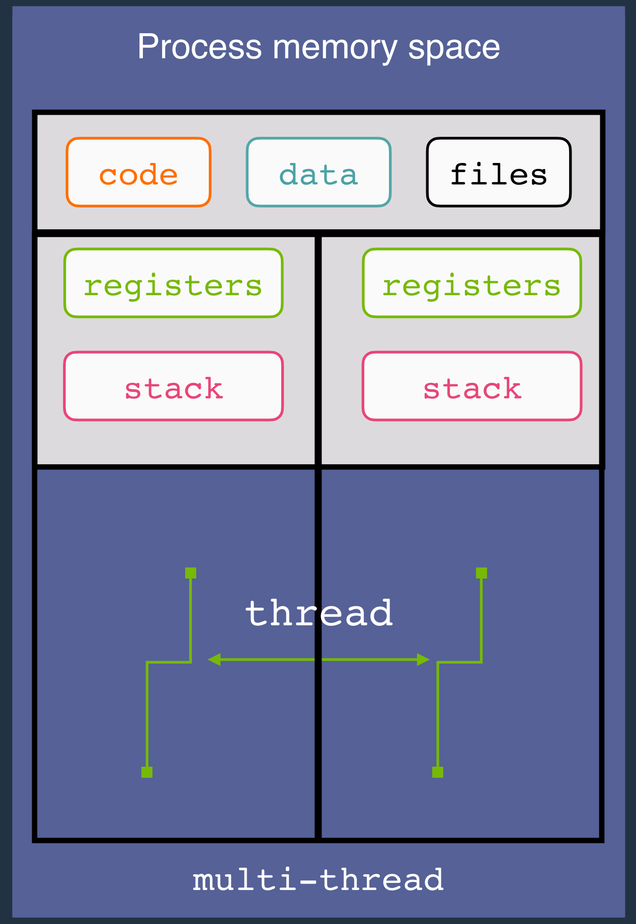
- A thread ID
- A program counter
- A register set
- A stack

The instructions executed by a thread can be accessed from:

- The process global memory (data)
- The thread local stack

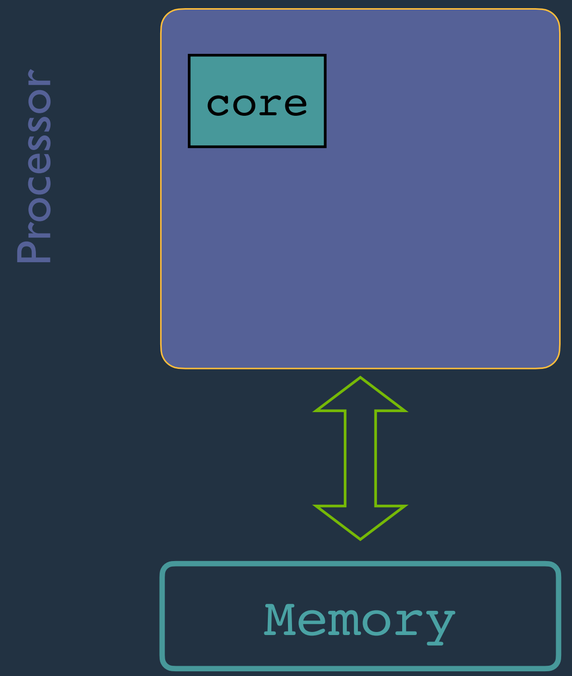
Each thread shares the same heap, data and resources

- A thread is a lightweight process that can be managed independently by a scheduler.
- It improves the performance using parallelism.

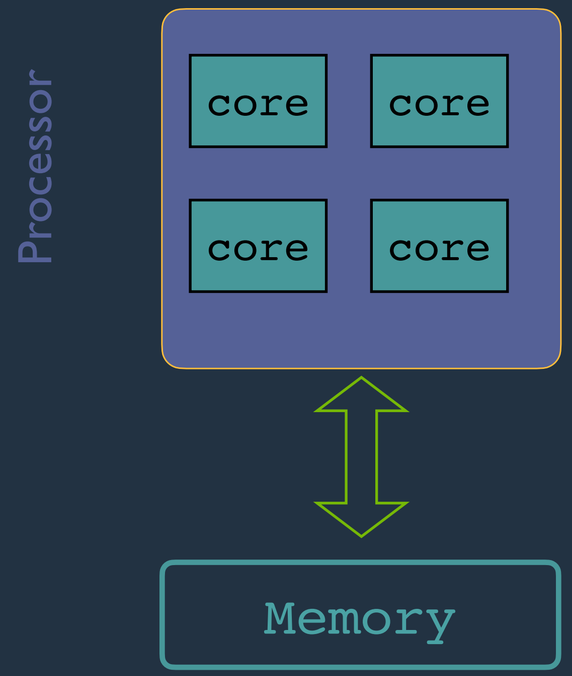


Sequential computing completes tasks one at time, orderly

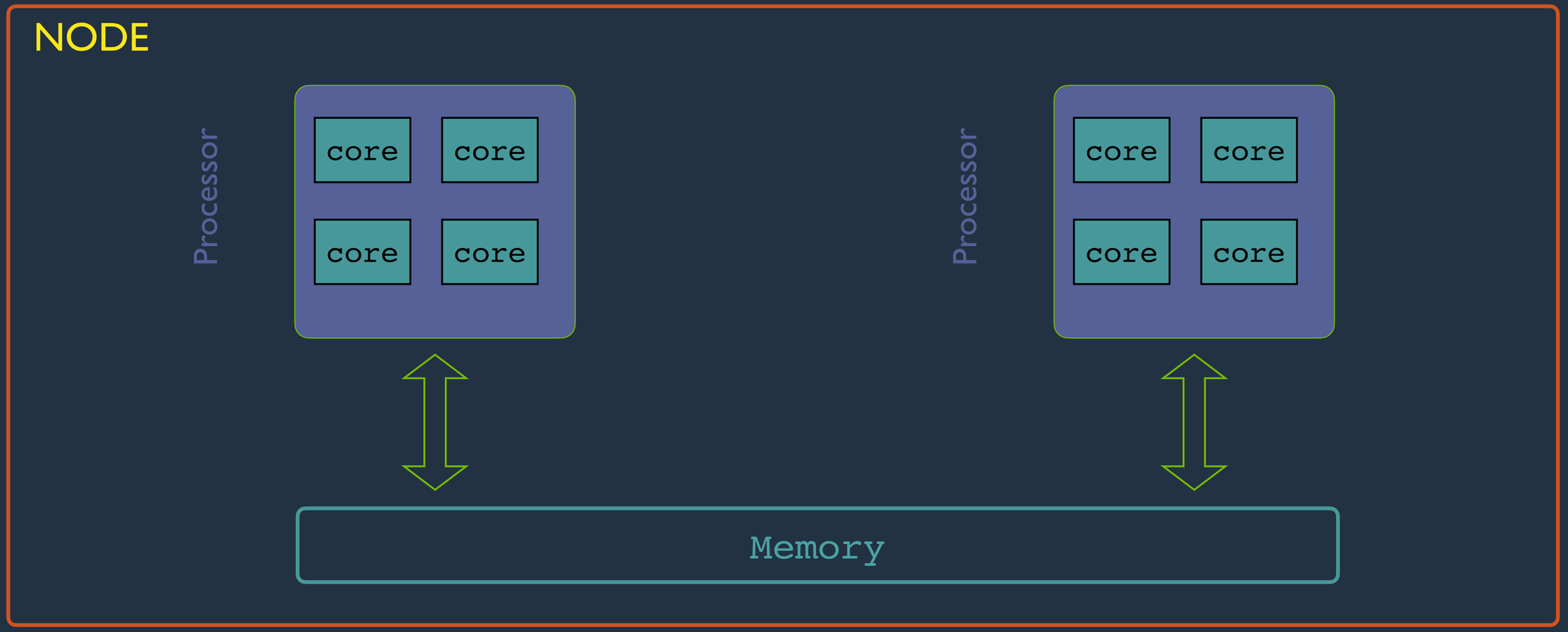
Older processor had only one cpu core to execute instructions



Modern processors have 4 or more independent cpu cores to execute instructions



A node consists of two processors (4 cpu cores each)



A node consists of two processors (4 cpu cores each)

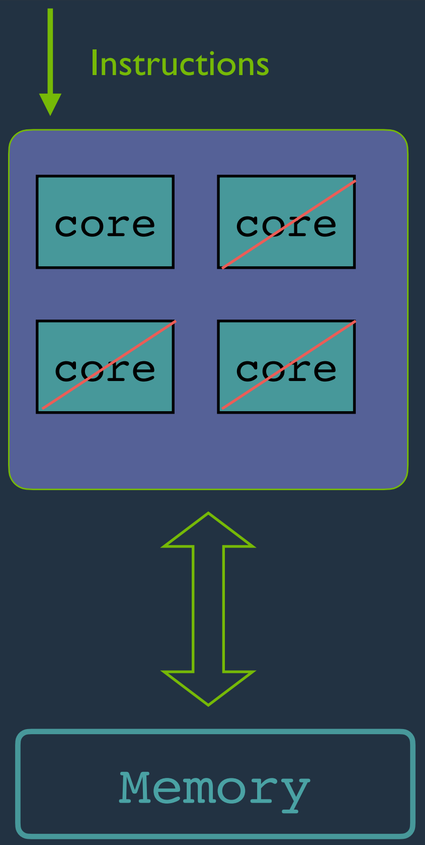
NODE

Sequential computing completes tasks one at time, orderly

- Instructions executed on 1 core
- Others cores idle

Waste of available resources. We want all cores to be used to execute program

How ?



Let's clear off some terminology: Cores Vs Threads

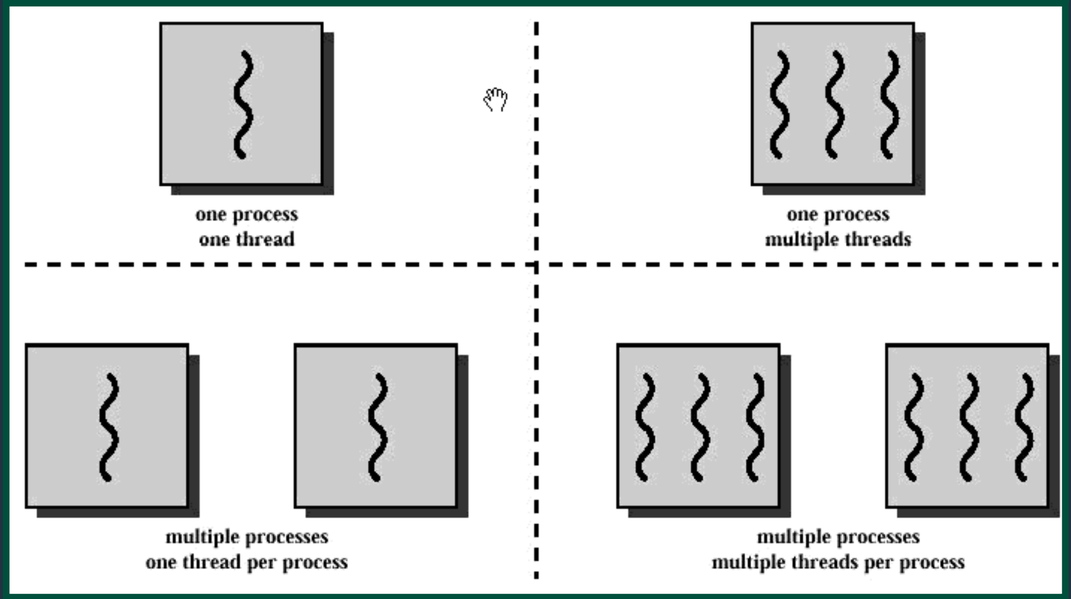
Core is a hardware construct

- A core is a single computing component with independent of CPUs
- It's a physical part of the CPU which read and execute program instructions
- They do all heavy lifting

Threads are software contract

- It is a set of sequential instructions that are executed in order
- Often each thread in a program is mapped to a single core

Modern computers are composed of multiple processors and multiple thread



Three prevailing types of parallelism

Modern systems exhibits at least three prevailing types of parallelism

Shared memory system

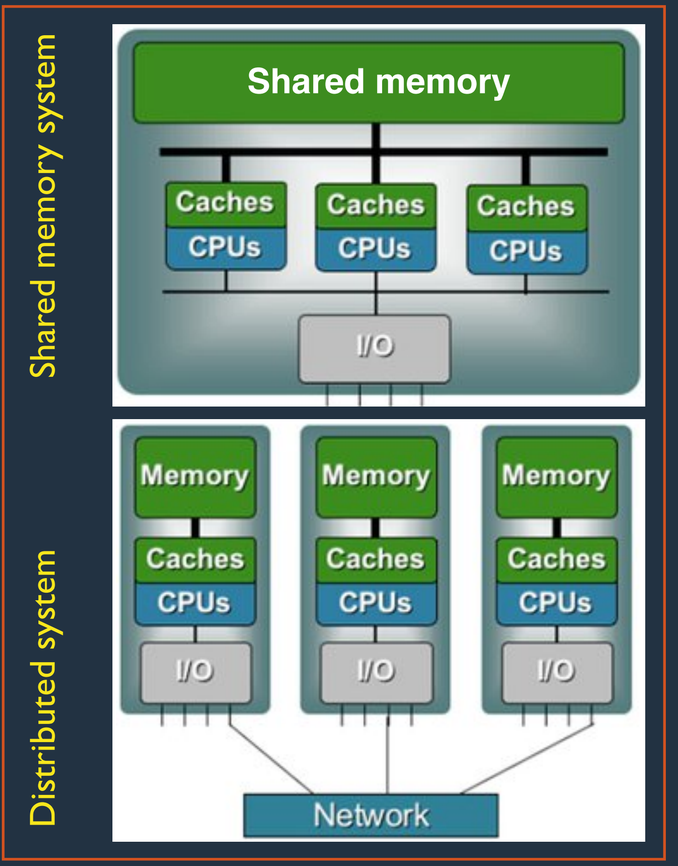
- Multiple processing units
- Shared memory

Distributed system

- Uses more than one computer
- Has its own processing unit and physical memory
- These computers are connected with fast interconnection networks

Graphic processor units (GPU):

- Used as co-processors for solving general purpose numerical intensive problems

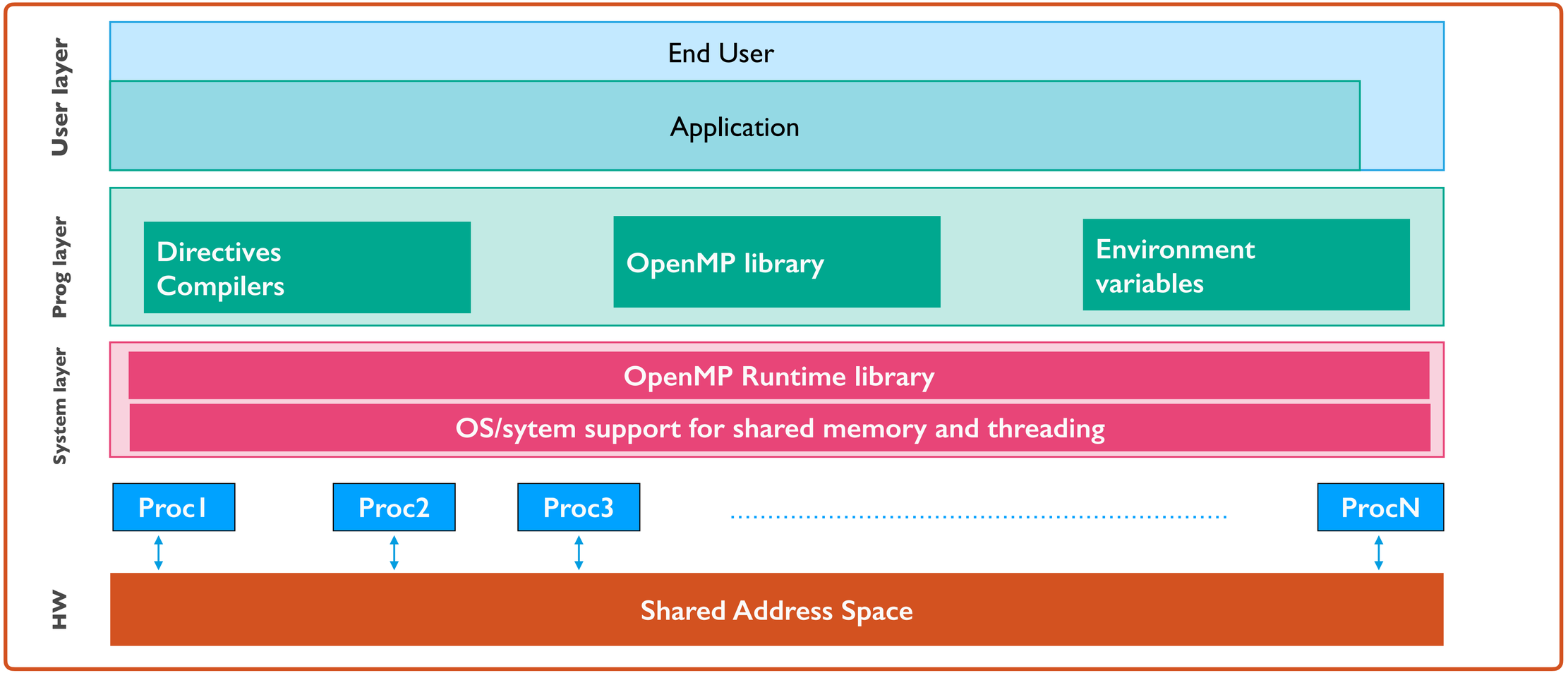


Here, I will talk only shared memory type of parallelism

Topics we will cover?

- Before starting parallel paradigms
- First steps with OpenMP
- Parallel construct
- Worksharing constructs
- Data sharing
- Synchronization Constructs

OpenMP is an API for writing multithreaded applications



Compiler directive apply to a structured block of statements

Directive based approach

- Directives are added to serial source code which manage the loop parallelization
- Open**MP** : Compiler directives, library routines, environment variables
- Easy to port of an original code both on many multicore CPUs and GPUs
- Works with C/C++ or Fortran

C/C++ OpenMP: parallel

```
#pragma omp construct [clause,  [, clause] …]{
  // line code
}
```

Fortran OpenMP: parallel

```
!$omp construct [clause,  [, clause] …]
  // line code
!$omp end parallel
```

Compiler directive apply to a structured block of statements

C/C++ OpenMP: parallel

```
#pragma omp parallel num_threads(4){
  // line code
}
```

Fortran OpenMP: parallel

```
!$omp parallel num_threads(4)
  // line code
!$omp end parallel
```

Library API calls

- Fortran:

  use omp_lib

  !$ print, "OpenMP support"*

- C/C++

  include <omp.h>

Conditional compilation

- Fortran:

  !$ print, "OpenMP support"*

- C/C++

```
#ifdef _OPENMP
  // code that requires OpenMP library functions
#endif
```

What do you need to start with?

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

GNU Fortran compiler OpenMP command line switch

No need to include the library if only using the compiler directives. The library only gets you the API calls.

Topics we will cover?

- Before starting parallel paradigms
- First steps with OpenMP
- Parallel construct
  - Creating threads with OpenMP
  - Setting number of threads
  - Dividing work between threads
  - First exercise
- Worksharing constructs
- Data sharing
- Synchronization Constructs

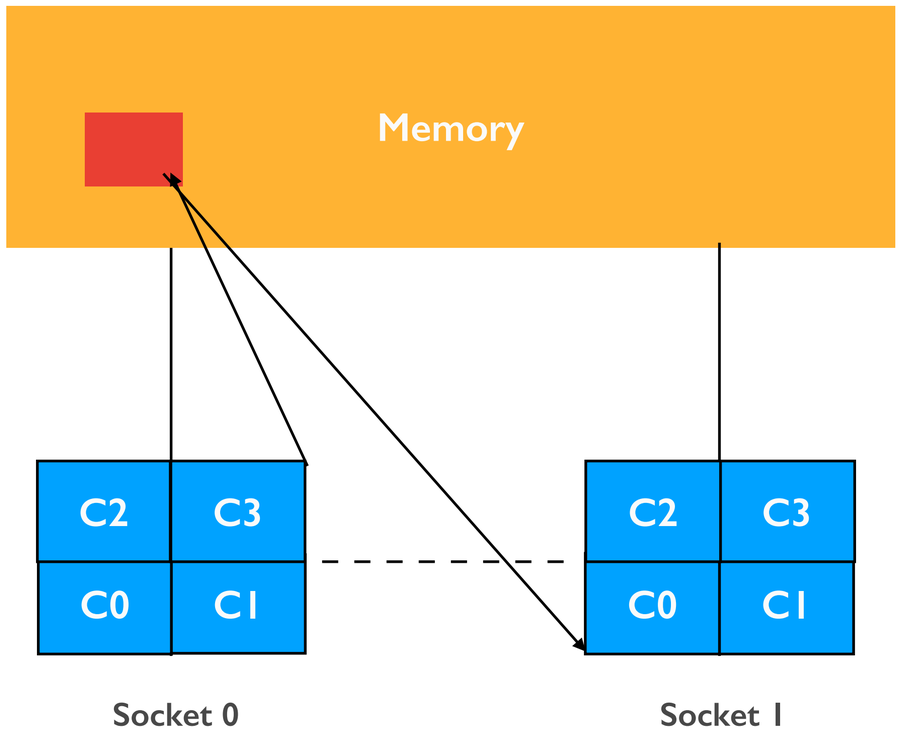
Perquisites of shared memory parallelisation

Shared memory programming

- A thread is a runtime entity that is able to independently execute a stream of instructions.
- All threads have access to a shared address space.
- It can read and write from the same memory.

Thread consists a set of sequential instructions that are executed in order

A typical HPC node consisting of 2 multi-core CPUs



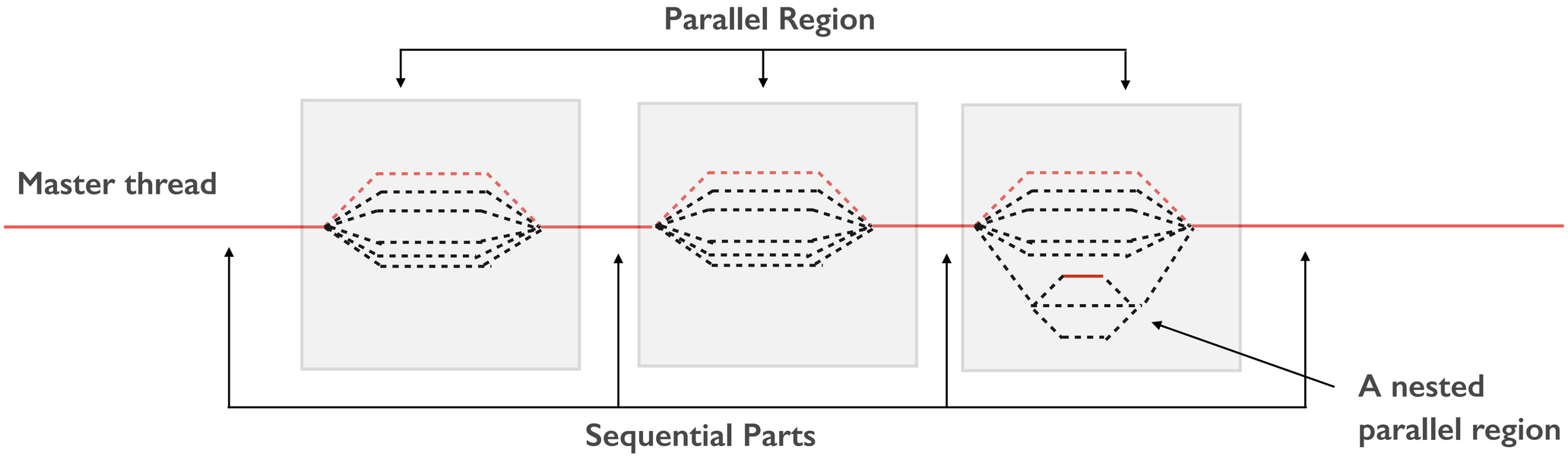
Perquisites of shared memory parallelisation

Fork-Join Parallelism

Fork : Master threads spawns a team of threads as needed

Parallelism added incrementally until performance goals are met

Join : At the end of the parallel region the thread team ends the execution and only the master thread continues the execution of the (serial) program



Sample example: Hello world

Fortran serial code

```
program hello
 !Serial code
  print*, “Hello world!”
 !Serial code
end program hello
```

C serial code

```
#include <stdio.h>
 int main()
 {
  // Serial code
   print(“Hello world! \n ”);
  // Serial code
 }
```

Parallel region creates a team of threads that execute the workload

Fortran code

```
program hello
 !$omp parallel
  print*, “Hello world!”
 !$omp end parallel
end program hello
```

C code

```
#include <stdio.h>
 int main()
 {
  #pragma omp parallel
  {
   printf(“Hello world! \n ”);
  }
 }
```

So what really happened?

- Threads redundantly execute code in the block
- Each thread will conduct Hello world!
- Threads are synchronised at the end of the parallel region

Number of threads used by OpenMP can be controlled

Three ways to set the number of threads

- By using environment variables:

```
# Set number of threads for the entire session
export OMP_NUM_THREADS = 16; ./program

# Set up for one execution of the program
OMP_NUM_THREADS = 16 ./program
```

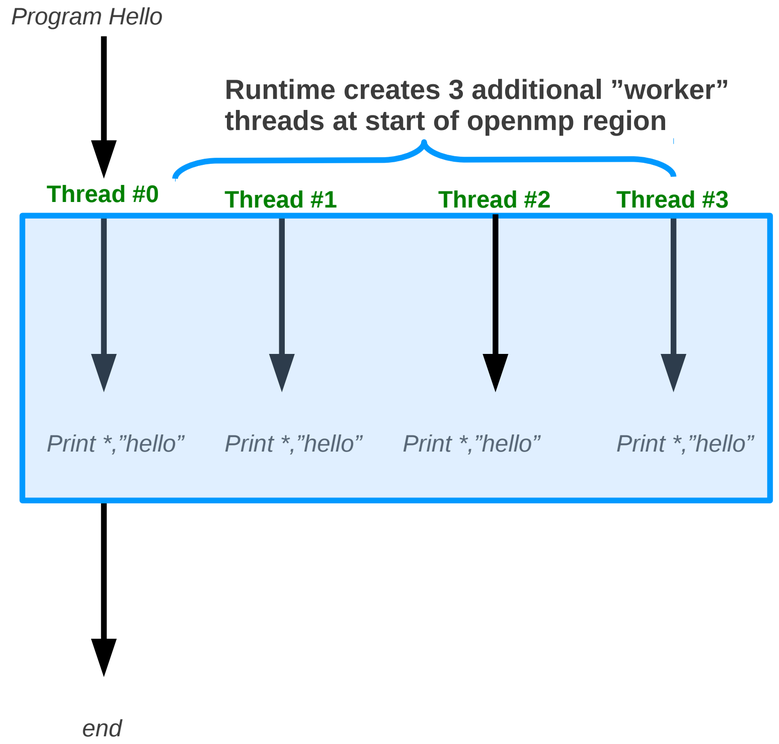
- By appending a clause to the OpenMP directive:

```
For Fortran code:     call omp_set_num_threads(16)
For C code :          #pragma omp parallel num_threads(16)
```

- Rule of thumb: number of threads = number of cores

At run time

Fortran serial code



OMP region, every thread executes all the instructions in the OMP block

```
$ ./output
Hello world !
Hello World !
Hello world !
Hello World !
```

Dividing work between threads

OpenMP API gives you calls to determine thread information when inside a parallel region

How many threads are there?

```
nthreads = omp_get_num_threads()
```

What is my thread index

```
tid = omp_get_thread_num()
```

Note: In OpenMP the threads starts from 0 …. num_threads-1

Setting number of threads

Fortran serial code

```
program hello
USE omp_lib
 IMPLICIT NONE
 INTEGER :: nthreads = 4
 CALL OMP_SET_NUM_THREADS(nthreads)
 INTEGER :: tid

 !$omp parallel
  tid=OMP_GET_THREAD_NUM()
  print*, “Hello world from = ”, tid, “with”, &
         OMP_GET_NUM_THREADS(), “threads”
 !$omp end parallel
 print *, “all done, with hopefully”, nthreads, “threads”

end program hello
```

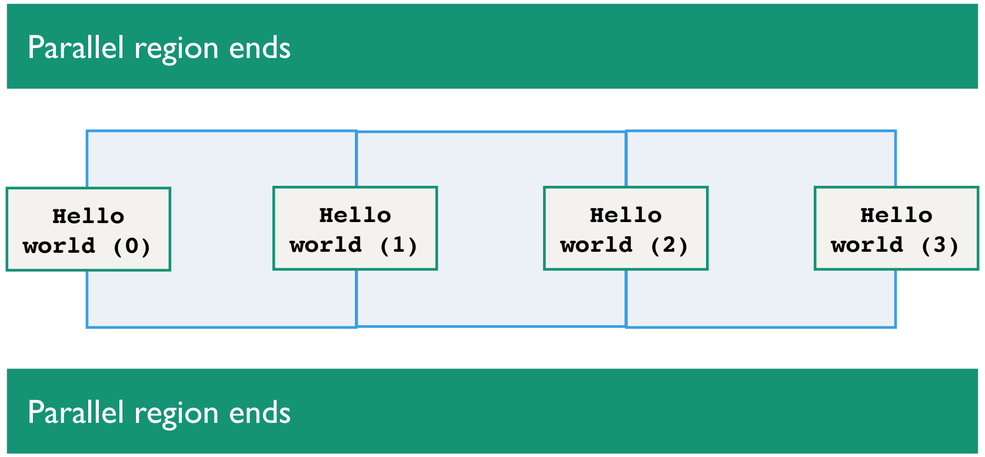
$./output

```
Hello world I’m tid   3  in nthreads  4
Hello world I’m tid   3  in nthreads  4
Hello world I’m tid   3  in nthreads  4
Hello world I’m tid   3  in nthreads  4
```

What is the problem here?

Each thread executed the same code redundantly

OMP_SET_NUM_THREADS(4)



Creating OpenMP threads

Fortran serial code

```
program hello
USE omp_lib
 IMPLICIT NONE
 INTEGER :: nthreads, tid

 !$omp parallel private(tid)
  tid=OMP_GET_THREAD_NUM()
  Nthreads = OMP_GET_NUM_THREADS()

  print*, “Hello world from thread = ”, tid , “with ”,
nthreads, “threads”
 !$omp end parallel

end program hello
```

$./output

```
Hello world from thread =  0 with  4  threads
Hello world from thread =  3 with  4  threads
Hello world from thread =  1 with  4  threads
Hello world from thread =  2 with  4  threads
```

In general, better to use environment variables, it is flexible at run time

First exercise

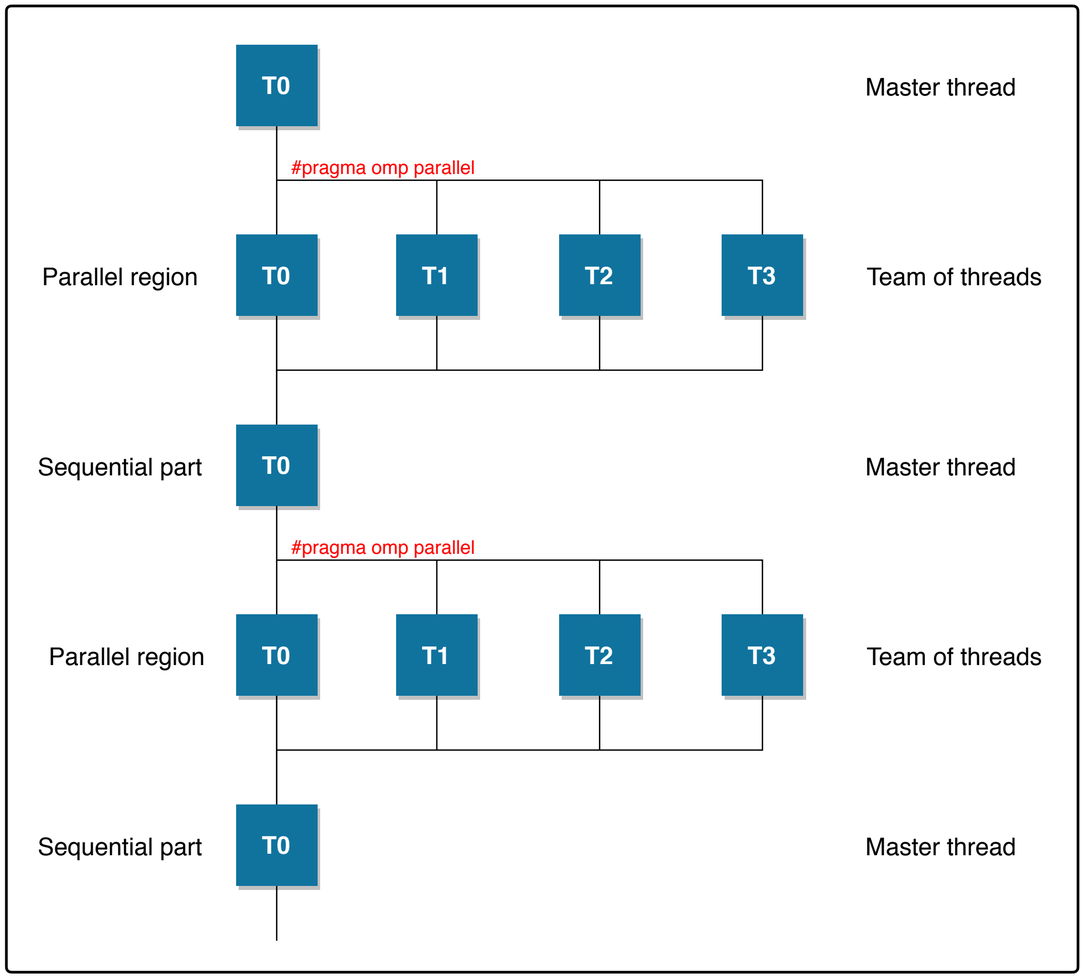
Vector addition code

a) Add a parallel loop
b) Get thread IDs
c) Compile
d) Run
e) Optional experiments with OMP_NUM_THREADS

Is this what you have expected?

```
program vecAdd
  IMPLICIT NONE
  real(kind=8)    ::  start, finish
  INTEGER         ::  N = 1e8
  real(kind=8), allocatable :: A(:), B(:), C(:)
  INTEGER         :: i
  ! Allocate memory
  allocate( A(N), B(N), C(N))
  ! Initialise data
  DO I = 1, N
  A(i) = 1
  B(I) = 2
  END DO
  ! Loop --
  Call cpu_time(start)
  DO I = 1, N
  C(I) = A(I) + B(I)
  END DO
  Call cpu_time(finish)
  ! Print result --
  write(*, “(A, F10.3)”) “RUNTIME : ”, finish-start
end program vecAdd
```

Recap



- Program starts as one single-threaded process
- Forks into teams of multiple threads when appropriate
- Stream of instructions might be different for each thread
- Information is exchanged via shared parts of memory

Vector add: serial code

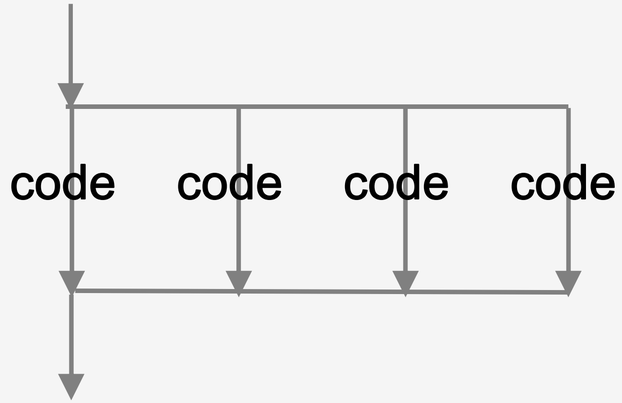
Add parallel region around work

```
Call cpu_time(start)

 DO I = 1, N
 C(I) = A(I) + B(I)
 END DO

Call cpu_time(finish)

! Print result --
 write(*, “(A, F10.3)”) “RUNTIME : ”, finish-start
```

Vector add: step 1

Add parallel region around work

```
Call cpu_time(start)
!$omp parallel
 DO I = 1, N
 C(I) = A(I) + B(I)
 END DO
!$omp end parallel
Call cpu_time(finish)
! Print result --
 write(*, “(A, F10.3)”) “RUNTIME : ”, finish-start
```



Every thread will do the entire vector addition- redundantly

Vector add: step 2

Get thread IDs

```
integer :: tid, nthreads
!$omp parallel
tid = omp_get_thread_num()
nthreads = omp_get_num_thread()
 DO I = 1, N
 C(I) = A(I) + B(I)
 END DO
!$omp end parallel
! Print result ––
 write(*, “(A, F10.3)”) “RUNTIME : ”, finish-start
```

Vector add: step 2

Get thread IDs

```
integer :: tid, nthreads
!$omp parallel
tid = omp_get_thread_num()
nthreads = omp_get_num_thread()
 DO I = 1, N
 C(I) = A(I) + B(I)
 END DO
!$omp end parallel
! Print result ––
 write(*, “(A, F10.3)”) “RUNTIME : ”, finish-start
```

Incorrect behaviour at runtime

What’s the problem here?

Vector add: step 3

Get thread IDs

```
integer :: tid, nthreads
!$omp parallel private(tid)
tid = omp_get_thread_num()
nthreads = omp_get_num_thread()
 DO I = 1, N
 C(I) = A(I) + B(I)
 END DO
!$omp end parallel
! Print result ––
 write(*, “(A, F10.3)”) “RUNTIME : ”, finish-start
```

- In Open**MP**, all variables are shared between threads
- Each thread has its own copy of tid
- Solution: use the private clause on the parallel region
- This gives each thread its own unique copy in memory for the variables

Worksharing in an Open**MP** program

Sharing the work among threads

Step-3: work sharing (manual approach)

Finally, distribute the iteration space across the threads

```
integer :: tid, nthreads
!$omp parallel
tid = omp_get_thread_num()
nthreads = omp_get_num_thread()
DO I = (1+ tid*N/nthreads), (tid +1 )*N/nthreads
 C(I) = A(I) + B(I)
END DO
!$omp end parallel
! Print result ––
 write(*, “(A, F10.3)”) “RUNTIME : ”, finish-start
```

Note

Thread IDs are numbered from 0 in OpenMP.
Be careful with your index calculation

How does the manual approach works?

```
DO I = (1+ tid*N/nthreads), (tid +1 )*N/nthreads
 C(I) = A(I) + B(I)
END DO
```

Suppose: N = 100 and nthreads = 4

N/nthreads = 25

tid = 0	tid = 1	tid = 2	tid = 3
1+ tid*N/nthreads = 1	1+ tid*N/nthreads = 26	1+ tid*N/nthreads = 51	1+ tid*N/nthreads = 76
(tid +1)*25 = 25	(1 + tid)*25 = 50	(tid +1)*25 = 75	(tid +1)*25 = 100
Thread ID 0 computes elements from index 1 to 25	Thread ID 1 computes elements from index 26 to 50	Thread ID 0 computes elements from index 51 to 75	Thread ID 1 computes elements from index 76 to 100

Topics we will cover?

- Before starting parallel paradigms
- First steps with OpenMP
- Parallel construct
- Worksharing constructs
  - An idea of worksharing
  - Loop constructs
  - The schedule clause
  - Single construct
  - Workshare construct (Fortran only)
  - Second exercise
- Data sharing
- Synchronization Constructs

SPMD vs Worksharing

A parallel construct by itself creates an SPMD or "Single Program Multiple data" program

each thread redundantly executes the same code

How do you split up pathways through the code between threads within a team?

This is called worksharing
A work sharing region has no barrier no entry
An implied barrier exists at the end, unless nowait is specified
Each region **must** be encountered by all threads or none
Every thread must encounter the **same sequence** of worksharing regions and barrier regions

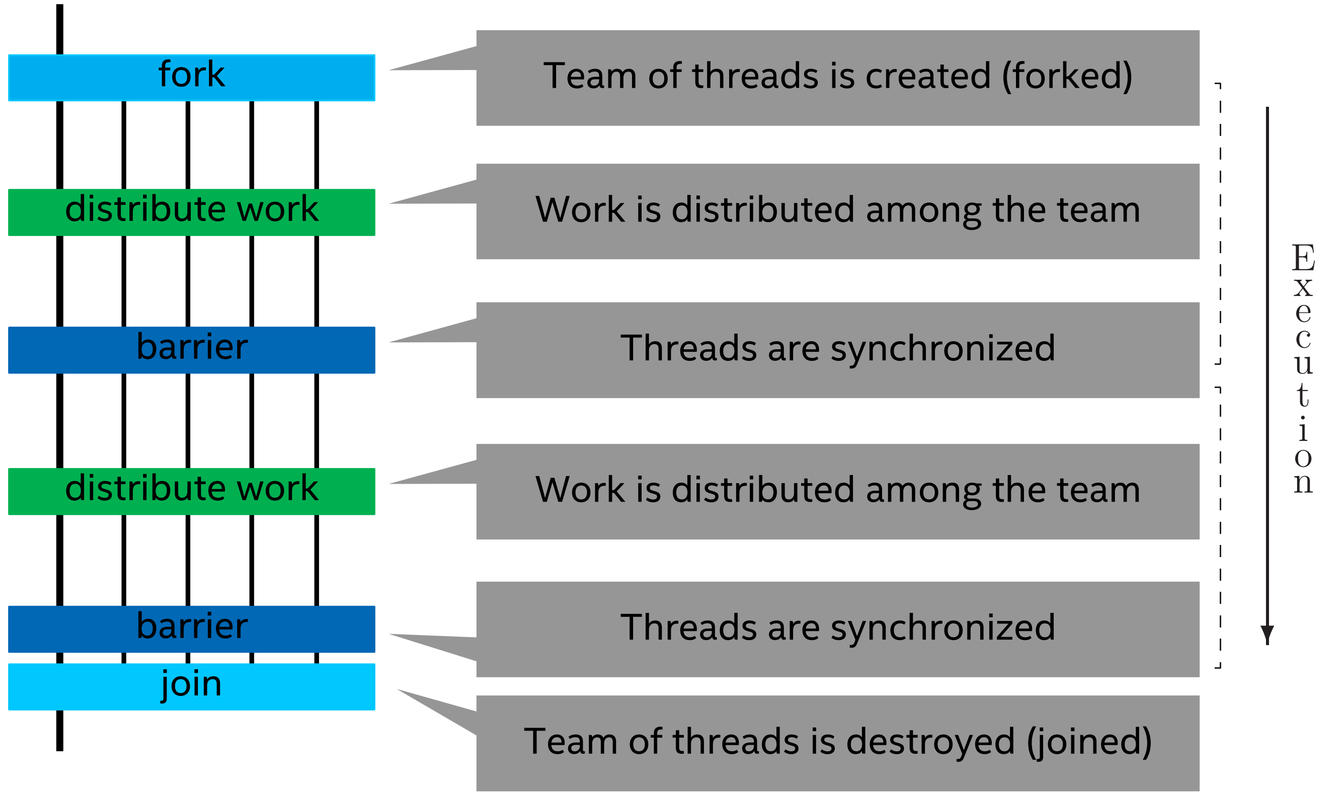
Commonly used worksharing directives in OpenMP

for/do loop
section
task construct
Workshare

Sharing the Work among threads

The loop iterator is made private by default: no need for data sharing clause

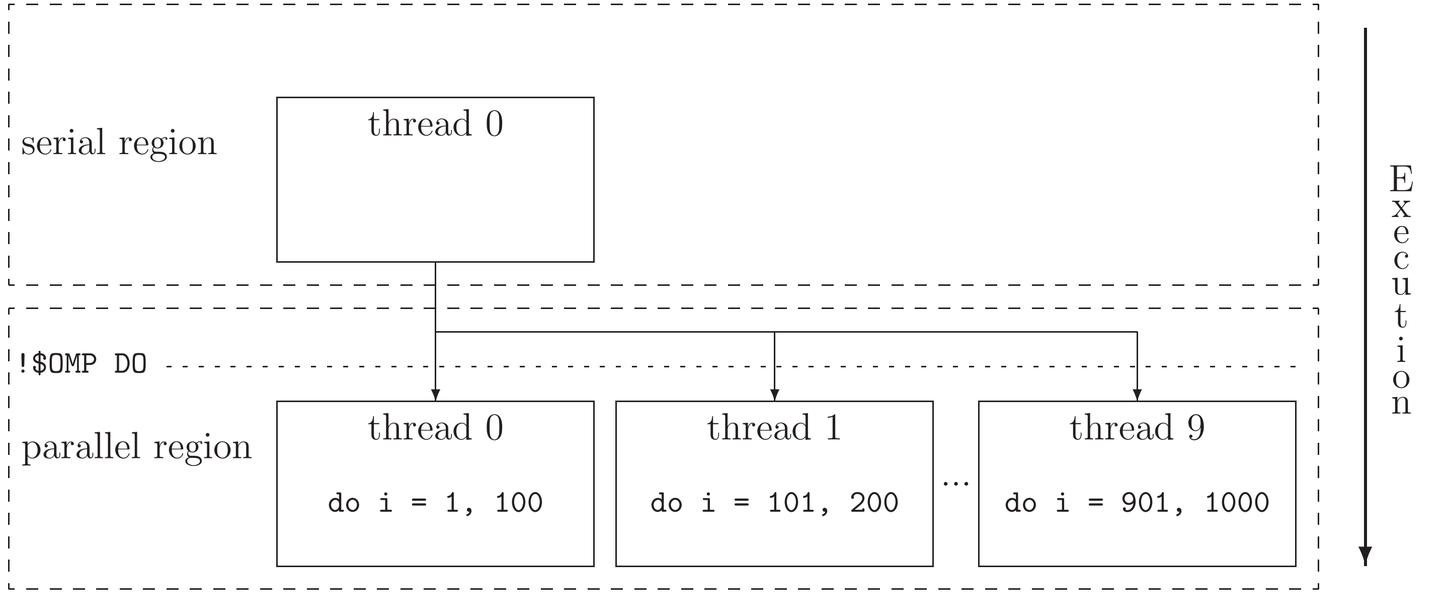
```
#pragma omp parallel
{
  #pragma omp for
   for (int i=0; i<N; i++){
     …
   }
   /* Implict barrier */
  #pragma omp for
  for (int i=0; i<N; i++){
    …
  }
  /* Implict barrier */
}
```



Sharing the Work among threads

The loop iterator is made private by default: no need for data sharing clause

```
!$omp parallel
!$omp do
DO I = 1, N
 C(I) = A(I) + B(I)
END DO
!$omp end do
!$omp end parallel
```



Combined worksharing directives (OpenMP approach)

The loop iterator is made private by default: no need for data sharing clause

```
1 !$omp parallel do
2 DO I = 1, N
3  C(I) = A(I) + B(I)
4 END DO
5 !$omp end parallel do
```

- Team of threads is formed at parallel region
- Loop iterations are split among threads, each loop iteration must be independent of other iterations
- After we finish our do loop there is an implicit sync point
- This is the unique part of the implicit behaviour of work sharing

Syntax of the loop construct in C/C++

#pragma omp for *[clause[[,] clause]...]*
`for-loop`

How OpenMP schedules loop iterations?

OpenMP provides different methods to divide iterations among threads, indicated by the schedule clause

Distribution of work: SCHEDULE clause

Schedule clause

- How the loop iteration are dived among the threads
- The form is specified by a chunk size

It can influenced by specifying the different schedule clause

- **Static**

  All iterations allocated to the threads before any iterations

  Low overhead and may exhibit high workload imbalance

- **Dynamic**

  Each thread is assigned some iteration at the beginning of loop execution

  High overhead and Can reduce workload imbalance

Chunks

- A set number of iterations
- Increasing chunk size reduces overhead and may increase cache hit rate
- Decreasing chunk size allows finer balancing of workloads

Different type

- `Static`
- `Dynamic`
- `Guided`
- `Auto`
- `Runtime`

Chunks are grouped of iterations that are assigned to threads

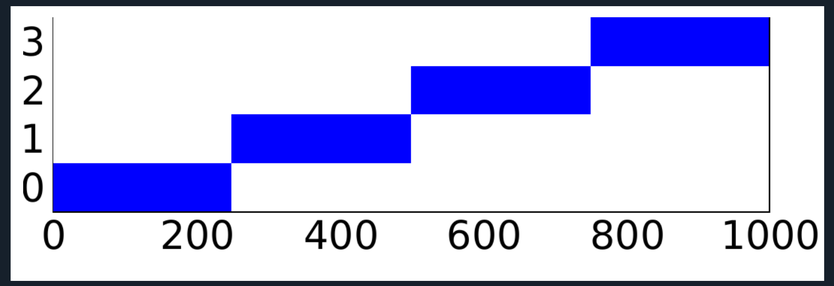
SCHEDULE(STATIC)

- Static schedule divides iterations into chunks and assigns chunks to threads in round-robin.
- If no chunk size specified, iteration space divided roughly equally.

```
!$omp parallel do private(num_threads) &
!$omp schedule(static, chunk)
do i = 1, N
    C(i) = A(i) + B(i)
end do
!$omp end parallel do
```

`schedule(static,16)`

Thread ID	Iterations
0	1–16, 65–80
1	17–32, 81–96
2	33–48, 97–100
3	49–64

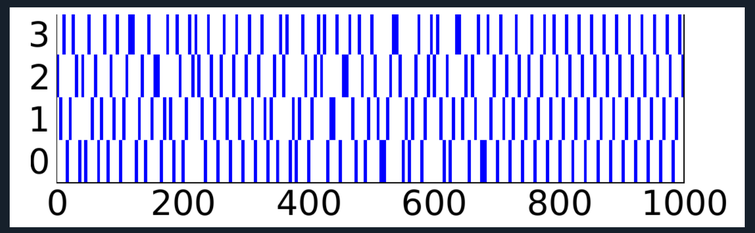


runtime: 4.197

SCHEDULE(Dynamic)

- Iteration space is divided into chunks according to chunk size.
- If no chunk size specified, default size is one.
- Each thread requests and executes a chunk, until no more chunks remain.
- Useful for unbalanced work-loads if some threads complete work faster.

```
!$omp parallel do private(num_threads) &
!$omp schedule(dynamic, chunk)
do i = 1, N
    C(i) = A(i) + B(i)
end do
!$omp end parallel do
```

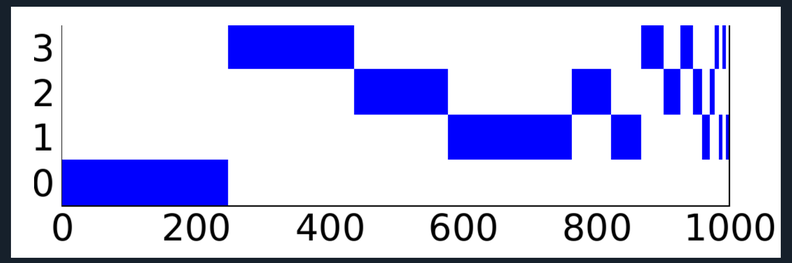


runtime: 3.747

SCHEDULE(Guide)

- Similar to assignment to dynamic, except the chunk size decreases over time.
- Granularity of work chunks gets finer over time.
- If no chunk size is specified, the default size is one.
- Useful to try to mitigate overheads of a dynamic schedule by starting with large chunks of work.

```
!$omp parallel do private(num_threads) &
!$omp schedule(guided, chunk)
do i = 1, N
    C(i) = A(i) + B(i)
end do
!$omp end parallel do
```



runtime: 0.136

Which is the best scheduler ?

Choosing the best schedule depends on understanding your loop

Second exercise

SAXPY: common operation in computations with vector processors included as part of the BLAS routines

$$y \leftarrow \alpha x + y$$

a) A combination of scalar multiplication and vector addition
b) Parallelize the following SAXPY code
c) Add work sharing construct
d) Experiments with schedules clause
e) Vary the number of threads using OMP_NUM_THREADS()

```
/* ..... initialize on the host ..... */

for ( i=0; i<ARRAY_SIZE; i++ )
    {
      x[i] = 1.0;
      y[i] = 2.0;
    }

 start_time = omp_get_wtime();
 for ( i=0; i<ARRAY_SIZE; i++ )
 {
   y[i] = scalar*x[i] + y[i];
 }
 end_time = omp_get_wtime();

/* ..... Terminate ..... */

  printf ( "\n" );
  printf("================================== \n");
  printf ("SAXPY Time: %f seconds \n", (end_time - start_time));
  printf("================================== \n");
```

Welcome back

Second exercise

Step-1

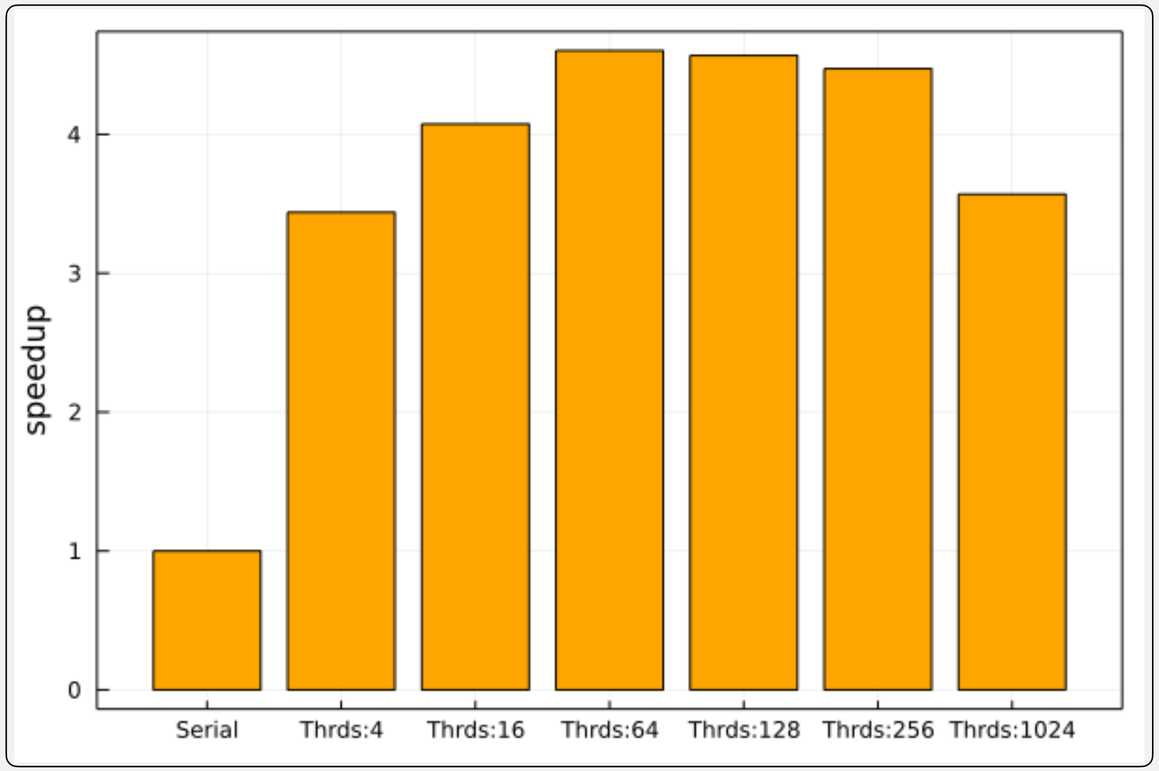
a) Threads perform some work
b) We add private clause
c) make code_c
d) ./binary.c
e) Compare the performance
f) Change the number of threads
   export OMP_NUM_THREADS=4, 16 ...

```
/* ......... initialize on the host ............. */
for ( i=0; i<ARRAY_SIZE; i++ )
    {
      x[i] = 1.0;
      y[i] = 2.0;
    }

 start_time = omp_get_wtime();
 #pragma omp parallel for private(i)
 for ( i=0; i<ARRAY_SIZE; i++ )
 {
   y[i] = scalar*x[i] + y[i];
 }
 end_time = omp_get_wtime();

/* ......... Terminate ............. */

  printf ( "\n" );
  printf("================================== \n");
  printf ("SAXPY Time: %f seconds \n", (end_time - start_time));
  printf("================================== \n");
```

Second exercise

Step-2

a) Threads perform some work
b) We add private clause
c) make code_c
d) export OMP_NUM_THREADS=32
e) ./binary.c



Speedup is limited (Amdahl's Law)

$$\text{Speedup (N)} = \frac{1}{(1-P) + \frac{P}{N}}$$

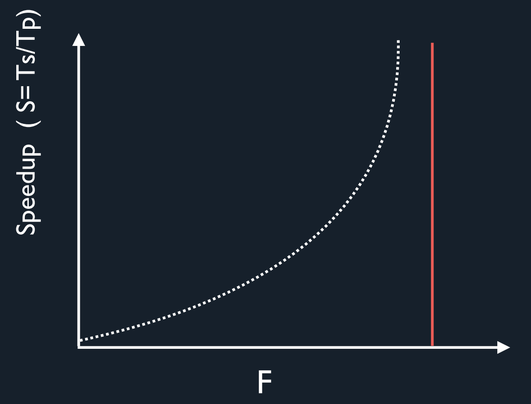
Parallel part is divided up by N workers

Serial part of job = 1 (100%) - parallel

Say 70% of a job can be made parallel and we use 2 processors.

P = 0.7 and N = 2

Speedup = 1.43



F = fraction of code that can be parallelized

OpenMP overhead/Scalability

- Starting a parallel OpenMP does not come free
- It is not always scale well

Explicit work sharing construct

- Sections
- Master
- Single

The section construct

Directive-pair allows to assign to each thread a completely different task leading to an MPMD* model of execution

Different blocks of work will be done by different threads

```
!$OMP PARALLEL
!$OMP SECTIONS
!$omp section
 // Block of code
!$omp section
 // Block of code
!$omp section
 // Block of code
!$OMP END SECTIONS
!$OMP END PARALLEL
```

Combined version


```
!$OMP PARALLEL SECTIONS
!$omp section
 // Block of code
!$omp section
 // Block of code
!$omp section
 // Block of code
!$OMP END PARALLEL
SECTIONS
```

Where/when to use:

- Pre-determined number of independent work units
- Ideally suited to call different function in parallel
- Used to set up a pipeline to overlap I/O with computations

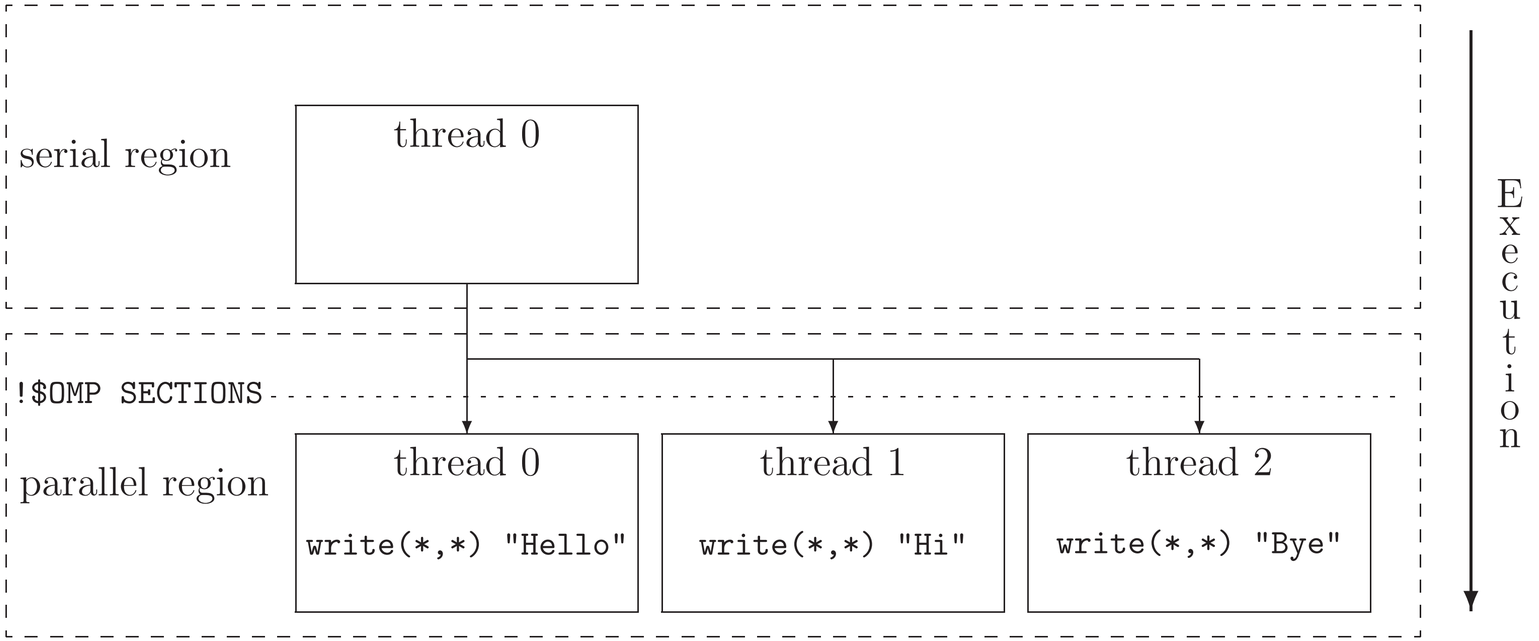
* Multiple Programs Multiple Data

The section construct

Directive-pair allows to assign to each thread a completely different task leading to an MPMD model of execution

Different blocks of work will be done by different threads

```
program hello_hi_bye

!$OMP PARALLEL SECTIONS
!$omp section
 write(*,*) "Hello"
!$omp section
 write(*,*) “Hi"
!$omp section
 write(*,*) “Bye”
!$OMP END PARALLEL SECTIONS

end program hello_hi_bye
```



The single construct

Execute a structured block with any one of a single thread

- First arrives to the opening-directive !$OMP SINGLE
- Executed by the first free thread
- **Implicit barrier** (unless a nowait clause is specified)
- **Optional clauses:**

  private

  First private

  Copy private

  No wait

C/C++

```
#pragma omp single [clause[[,] clause]...]
  // structured-block
```

Fortran

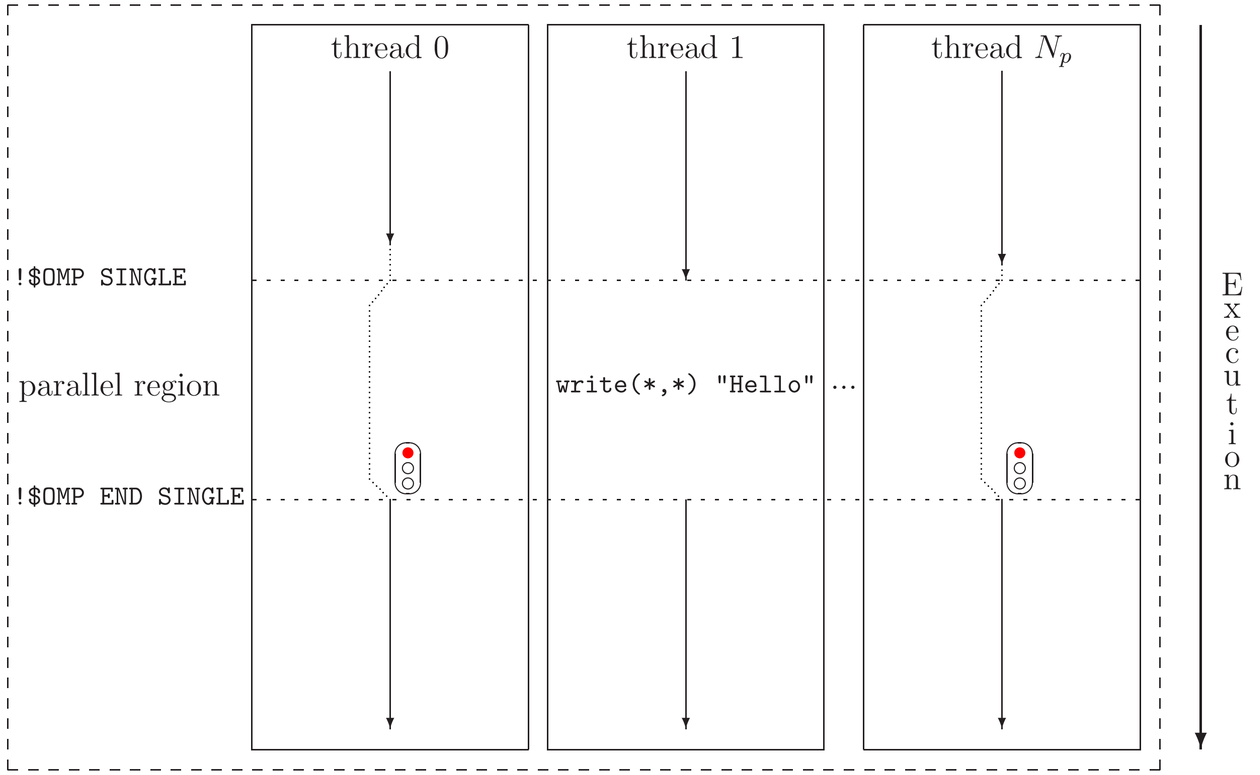
```
!$omp single [clause[[,] clause] ... ]
   ! structured-block
!$omp end single [nowait]
```

The single construct

Directive-pair allows to assign to each thread a completely different task leading to an MPMD model of execution

Execute a structured block with any one of a single thread

```
!$OMP SINGLE
  write(*,*) "Hello"
!$OMP END SINGLE
```



Specifies a block of code to be excited on a single thread

```c
#include <stdio.h>
void work1() {}
void work2() {}
int main(void)
{
  #pragma omp parallel num_threads(4)
  {
    int tid;
    tid = omp_get_thread_num();

    #pragma omp single
      printf("Beginning work1 on thread %d.\n", tid);

    work1();

    #pragma omp single
      printf("Finishing work1 on thread %d.\n", tid);

    #pragma omp single nowait
      printf("Finished work1 and beginning work2 on thread %d.\n", tid);

    work2();

  }
}
```

Build and run it:

```
$ gcc -fopenmp -o binary singleconst.c

$ ./binary
Beginning work1 on thread 0.
Finishing work1 on thread 3.
Finished work1 and beginning work2 on thread 1.

nshukla1 at login02 in /m100_scratch/userinternal/
$ ./binary
Beginning work1 on thread 0.
Finishing work1 on thread 1.
Finished work1 and beginning work2 on thread 2.

nshukla1 at login02 in /m100_scratch/userinternal/
$ ./binary
Beginning work1 on thread 0.
Finishing work1 on thread 0.
Finished work1 and beginning work2 on thread 1.
```

Where do we use them

- This directive must be used in parallel regions to access disk devices (i.e. open, read, write files) or may be used to update shared variables.
- Useful when dealing with operation that are not thread-safe
- Used to tracking simulation time
- Used for various book-keeping tasks which require thread-safe exectuion

The master construct

Execute a structured block master thread

Similar but specifies a structured block

- The code enclosed inside this directive-pair is executed only by the master thread of the team
- All the other threads continue with their work: no implied synchronization exists
- Closing-directive may give rise to a data race

There is one important difference. In contrast with the single construct, the master construct does not have an implied barrier on exit.

C/C++

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

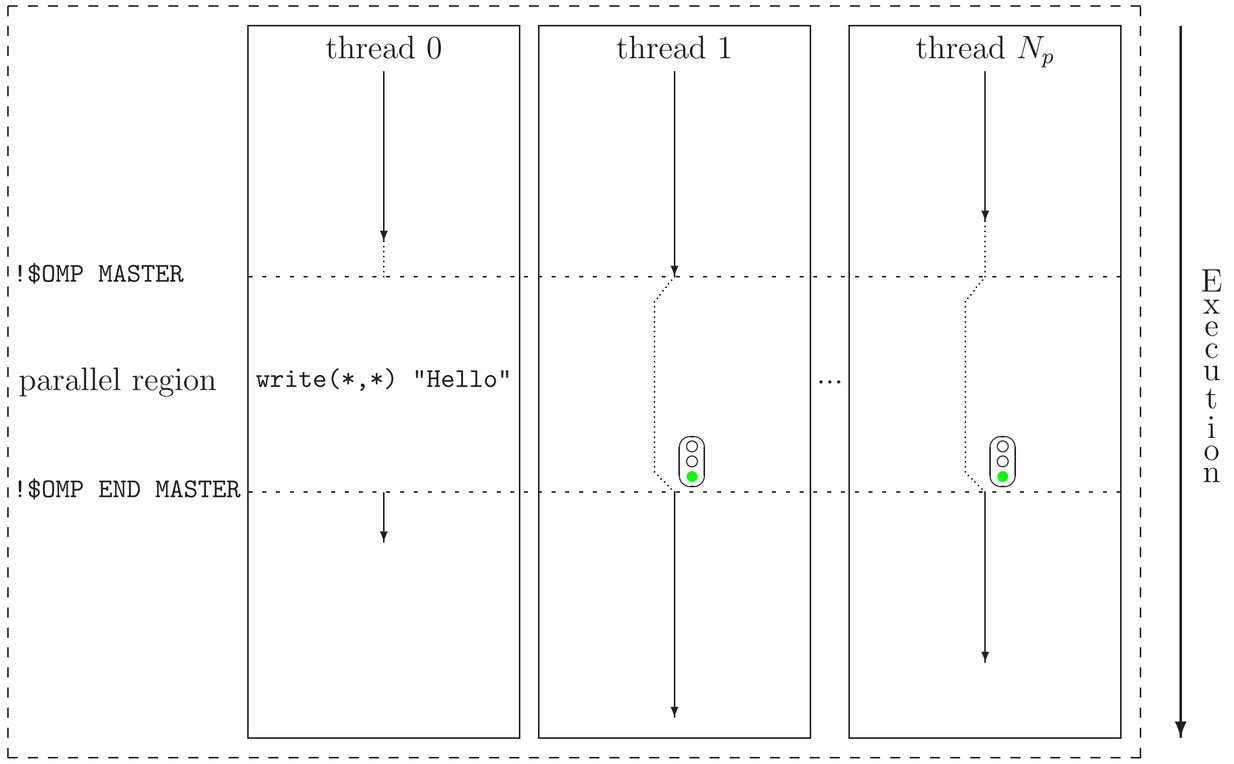
Fortran

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

The master construct

Execute a structured block master thread

```
!$OMP MASTER
  write(*,*) "Hello"
!$OMP END MASTER
```



Let's look at other example

What about nested loop?

Nested loop

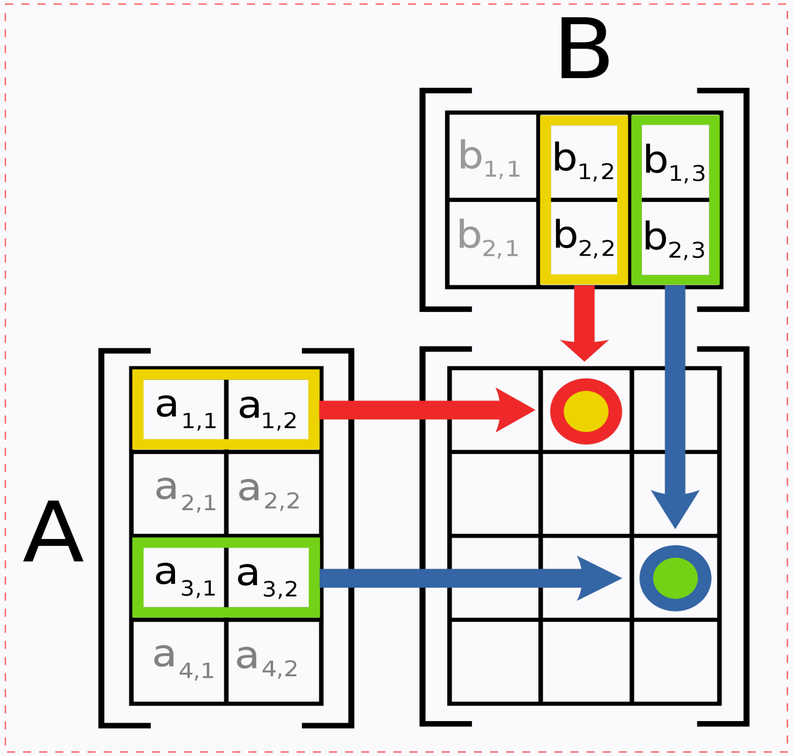
The loop iterator is made private by default: no need for data sharing clause

- 2D grid code, every cell is independent
- OpenMP worksharing would only parallelise over first loop with each thread performing inner loop serial
- Use the collapse(…) clause to combine iteration spaces
- OpenMP then wobrkshares the combined iteration space
- All N2 iterations are distributed across threads, rather than just the N of the outer loop

```
!$omp parallel do collapse(2)

DO I = 1, N
 DO J = 1, N
   …! loop body
 END DO
END DO

!$omp end parallel do
```

Third exercise: Matrix Multiplication

- Most Computational code involve matrix operations such as matrix multiplication
- Consider a matrix **C** of two matrices **A** and **B**
- Element *i,j* of **C** is the dot product of the *i*th row of **A** and *j*th column of **B**



Third exercise: Matrix Multiplication

Fortran Code

```
! Initilise data
! Optional
  do i = 1, M
     do j = 1, N
      A(i,j) = 2.0_8
      B(i,j) = 2.0_8
     end do
  end do
  do i = 1, M
    do j = 1, N
       C(i,j) = A(i,j) * B(i,j)
    end do
  end do
```

C/C++ Code

```
void  initMatrix(double *A, int n, int m, double c)
{
  int i,j;
  for (i=0; i<n; i++)
    for (j=0; j<m; j++)
      A[i*m+j] = c;
}
# Initilise data
  initMatrix((double *) A, N, M, 2.0);
  initMatrix((double *) B, N, M, 2.0);
  initMatrix((double *) C, N, M, 0.0);
  for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      C[i][j] = A[i][j] * B[i][j];
```

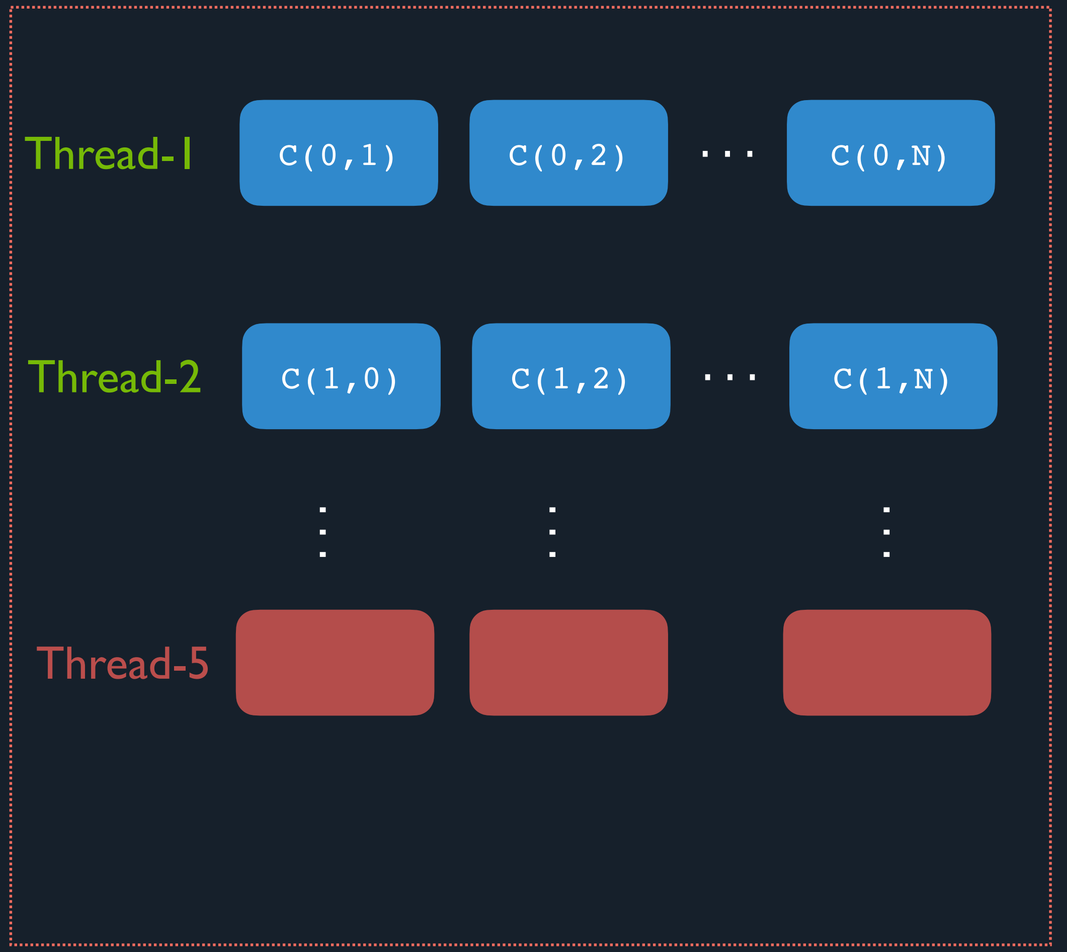
Welcome back

Third solution: Matrix Multiplication

Step-1

```
!$omp parallel do
 do i = 1, M
   do j = 1, N
      C(i,j) = A(i,j) * B(i,j)
   end do
 end do
!$omp end parallel do
```

- OpenMP threads are created
- Each thread computes its assigned portion of iteration space
- Threads synchronise and join

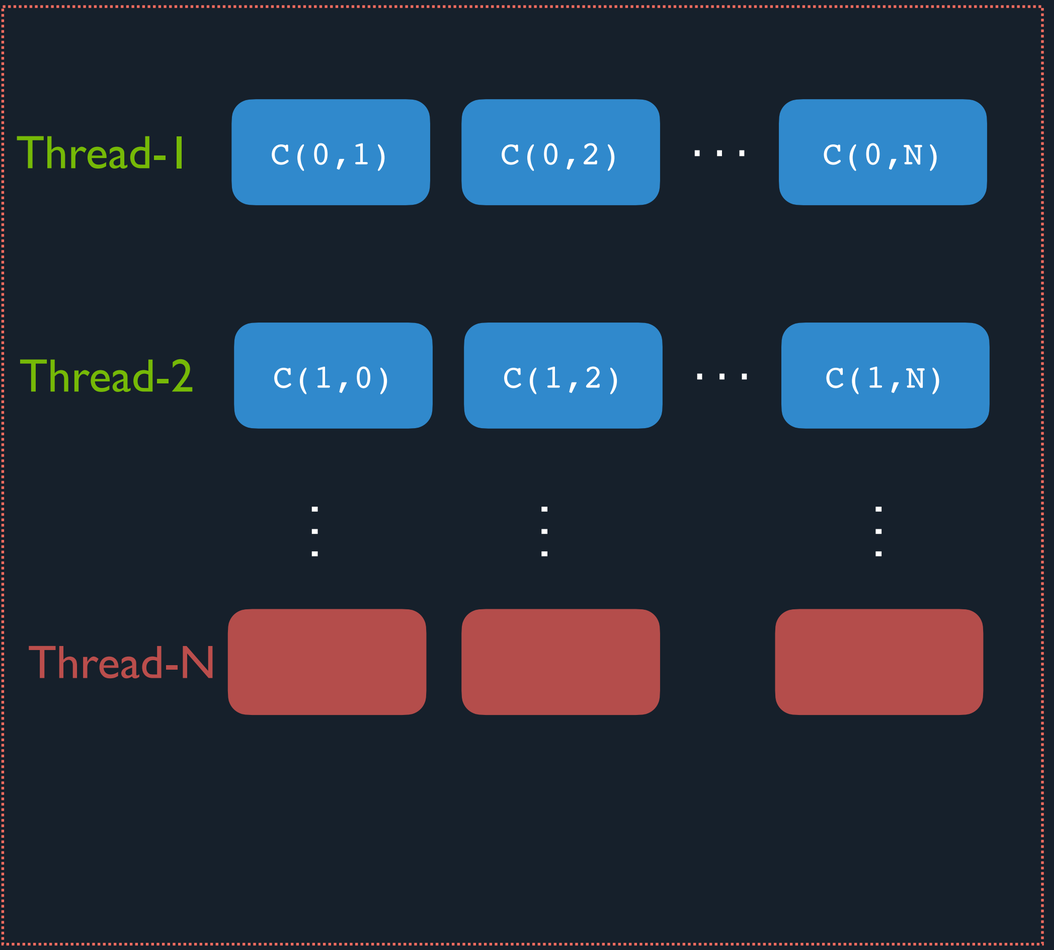


Third solution: Matrix Multiplication

Step-2

```
!$omp parallel do
    do i = 1, M
    !$omp parallel do
    do j = 1, N
        C(i,j) = A(i,j) * B(i,j)
      end do
    !$omp end parallel do
    end do
!$omp end parallel do
```

- "Nested parallelism" is disabled in OpenMP by default
- The second OMP is ignored at the runtime
- A team of only one thread is created
- Each inner loop is processed by a team of one thread
- More overhead in the inner loop.

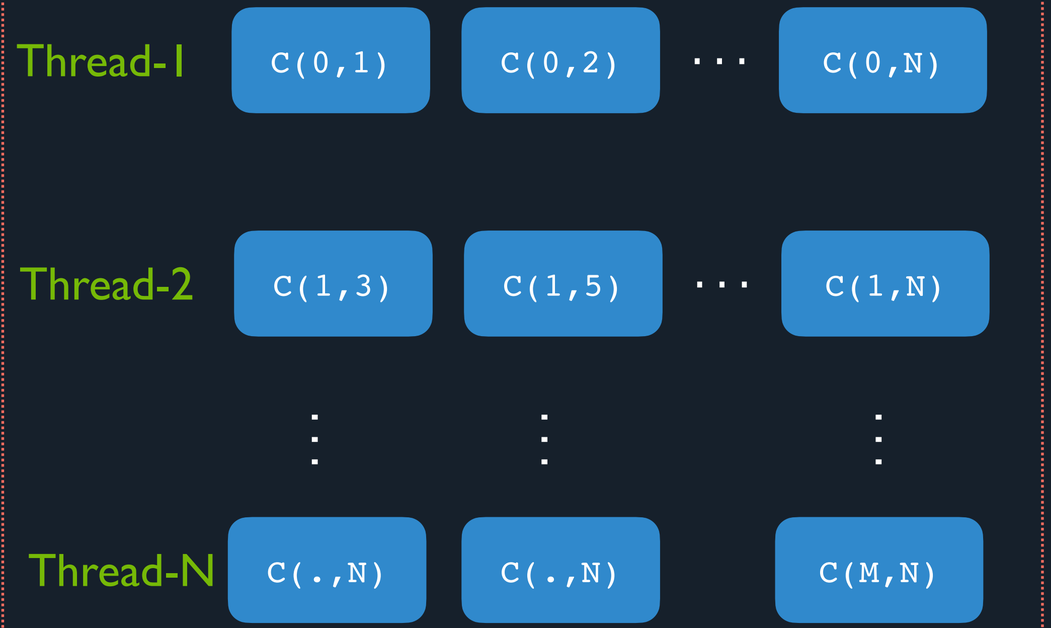


Collapse construct divides the work evenly

Step-3

```
!$omp parallel do collapse(2)
 do i = 1, M
   do j = 1, N
      C(i,j) = A(i,j) * B(i,j)
   end do
 end do
!$omp end parallel do
```

- OpenMP threads are created
- Loops are collapsed and iterations shared between threads
- Each thread computes its assigned portion of iteration space
- Threads synchronise and join



Topics we will cover?

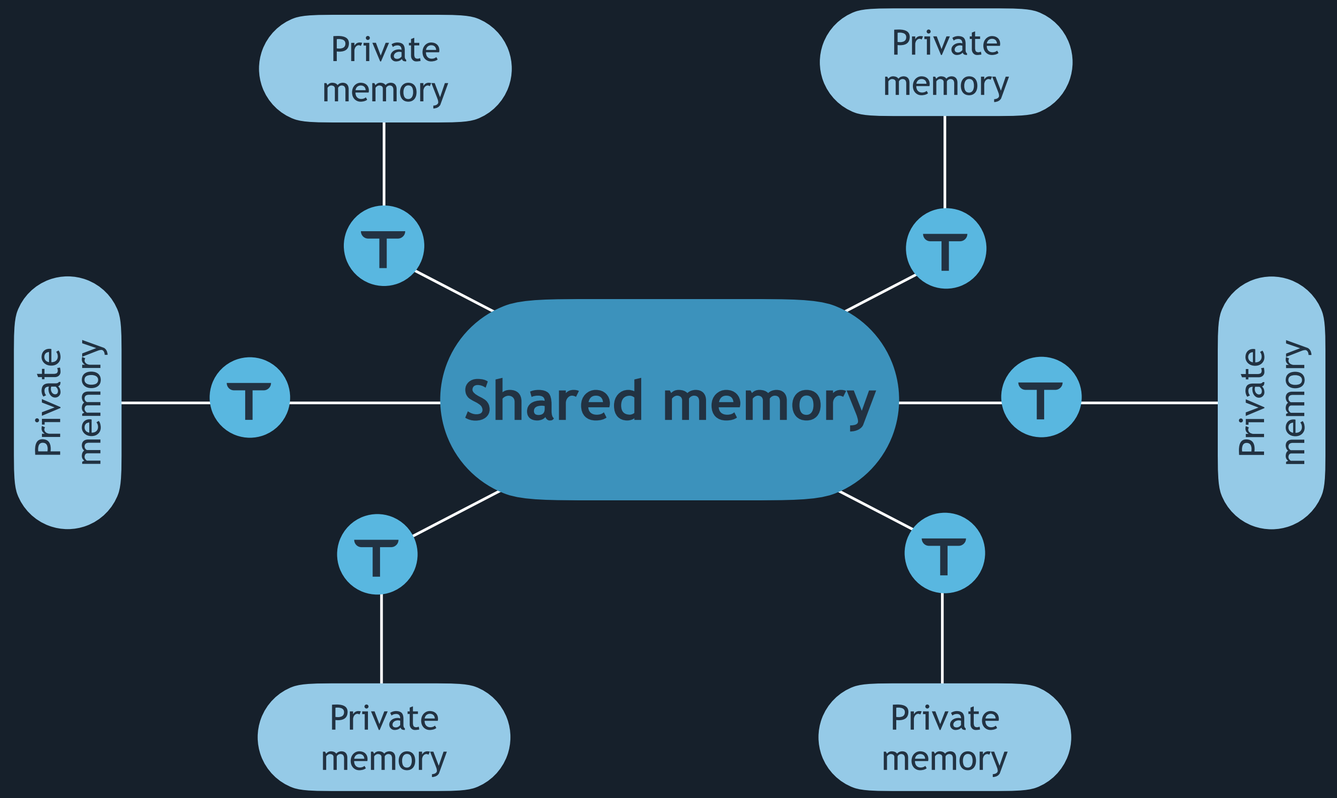
- Before starting parallel paradigms
- First steps with OpenMP
- Parallel construct
- Worksharing constructs
- Managing Data
  - Shared Clause
  - Private Clause
  - Firstprivate clause
  - Last clause
  - Tasks
- Synchronization Constructs

The OpenMP memory model

OpenMP program has two different elementary types of memory

- Private memory: Each thread has unique access to its private memory
- Shared memory: Each thread has access to the same single shared memory

Programmer must assign the appropriate memory types to the variables



OpenMP data scoping

Each thread is allowed to have its own temporary view of memory

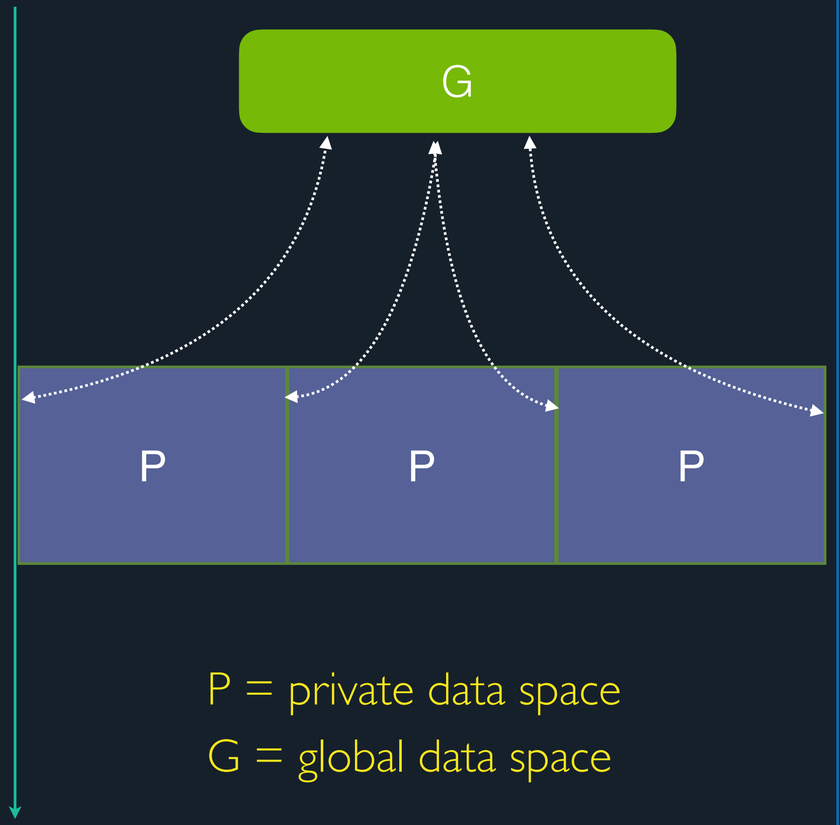
- Data-scoping clauses control how variables are shared within a parallel construct
- Includes

  `shared, private, firstprivate, lastprivate, reduction`

- Default variable scope:

  Variables are shared by default

  Global variables are shared by default



shared clause

Variable in the global data space are accessed by all the thread

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

int main(){
        int x;
        x = 3;

      #pragma omp parallel
       {
         x = x + 1;
         printf("Thread number: %d   shared: x is %d\n", omp_get_thread_num(), x);
       }
}
```

```
$ ./binary
Thread number: 5   shared: x is 6
Thread number: 7   shared: x is 7
Thread number: 0   shared: x is 6
Thread number: 3   shared: x is 6
Thread number: 1   shared: x is 6
Thread number: 2   shared: x is 6
Thread number: 4   shared: x is 7
Thread number: 6   shared: x is 7
```

private clause- direct compiler to make one more variables private

Each copy is only visible to its associated thread

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

int main(){
        int i, x;
        x = 3;

       #pragma omp parallel for private(x)
        for (i = 0; i < 10; i++) {
          x = i;
          printf("Thread number: %d   x: %d\n", omp_get_thread_num(), x);
        }
        printf("x is %d\n", x);
}
```

```
$ ./binary
Thread number: 6   x: 8
Thread number: 3   x: 5
Thread number: 0   x: 0
Thread number: 0   x: 1
Thread number: 7   x: 9
Thread number: 1   x: 2
Thread number: 1   x: 3
Thread number: 4   x: 6
Thread number: 5   x: 7
Thread number: 2   x: 4
x is 3
```

Inside a parallel region the scope of a variable can be shared or private

Several variations are available in OpenMP, which depends on the initial values and whether the results are copied outside the region

firstprivate(x)

- Each thread gets its own x variable
- It is initialised to the value of the original variable entering the region.
- The value of the original variables is undefined on region exit

lasprivate(x)

- Used for loops
- Each thread gets its own x variable, and on existing region the original variable is updated taking the value from the sequentially last iterations

There is also the threadprivate(x)

firstprivate clause

Compiler creates a private variables having an initial values identical to the value of the variable which is controlled by the master thread

```c
int main(){
        int i, x;
        x = 5;
        omp_set_num_threads(3);
       #pragma omp parallel for firstprivate(x)
        for (i = 0; i < 10; i++) {
          printf("Thread number: %d   x: %d\n", omp_get_thread_num(), x);
          x = i;
          printf("Thread number: %d   x: %d\n", omp_get_thread_num(), x);
        }
        printf("x is %d\n", x);
}
```

```
$outout

  Thread number: 1   x: 5
  Thread number: 1   x: 4
  Thread number: 1   x: 4
  Thread number: 1   x: 5
  Thread number: 1   x: 5
  Thread number: 1   x: 6
  Thread number: 2   x: 5
  Thread number: 2   x: 7
  Thread number: 2   x: 7
  Thread number: 2   x: 8
  Thread number: 2   x: 8
  Thread number: 2   x: 9
  Thread number: 0   x: 5
  Thread number: 0   x: 0
  Thread number: 0   x: 0
  Thread number: 0   x: 1
  Thread number: 0   x: 1
  Thread number: 0   x: 2
  Thread number: 0   x: 2
  Thread number: 0   x: 3
  x is 5
```

lastprivate clause

Used to copy back to the master thread's copy of variable the private copy of the variable from the thread that execute the sequentially last iteration

```c
int main(){
        int i, x;
        x = 5;
        omp_set_num_threads(3);
        #pragma omp parallel for lastprivate(x)
        for (i = 0; i < 10; i++) {
          x = i;
          printf("Thread number: %d   x: %d\n", omp_get_thread_num(), x);
        }
        printf("x is %d\n", x);
}
```

```
$outout

Thread number: 0   x: 0
Thread number: 0   x: 1
Thread number: 0   x: 2
Thread number: 0   x: 3
Thread number: 2   x: 7
Thread number: 2   x: 8
Thread number: 2   x: 9
Thread number: 1   x: 4
Thread number: 1   x: 5
Thread number: 1   x: 6
x is 9
```

Topics we will cover?

- Before starting parallel paradigms
- First steps with OpenMP
- Parallel construct
- Worksharing constructs
- Data sharing
- Synchronization Constructs
  - High level synchronization
  - Low level synchronization

Sum of all integers from 1 to n-1

```
program race_condition
 USE omp_lib
 IMPLICIT NONE
 real(kind=8) :: start, finish
 integer i, n, sum
 n = 10000
 sum = 0
 Call cpu_time(start)

 DO I = 1, n-1
  sum = sum + I
 END DO

 Call cpu_time(finish)
 ! Print result --
  write(*, "The total sum : ", sum, ((n-1)*n)/2
  write(*, "(A, F18.9)") "RUNTIME : ", finish-start
end program race_condition
```

Sample output

```
The total sum :  704982704     704982704
      Runtime :  0.00062800
```

Sum of all integers from 1 to n-1

```
program race_condition
 USE omp_lib
 IMPLICIT NONE
 real(kind=8) :: start, finish
 integer i, n, sum
 n = 10000
 sum = 0
 Call cpu_time(start)
 !omp parallel do
DO I = 1, n-1
 sum = sum + I
END DO
 !omp end parallel do
 Call cpu_time(finish)
 ! Print result --
  write(*, "The total sum : ", sum, ((n-1)*n)/2
  write(*, "(A, F18.9)") "RUNTIME : ", finish-start
end program race_condition
```

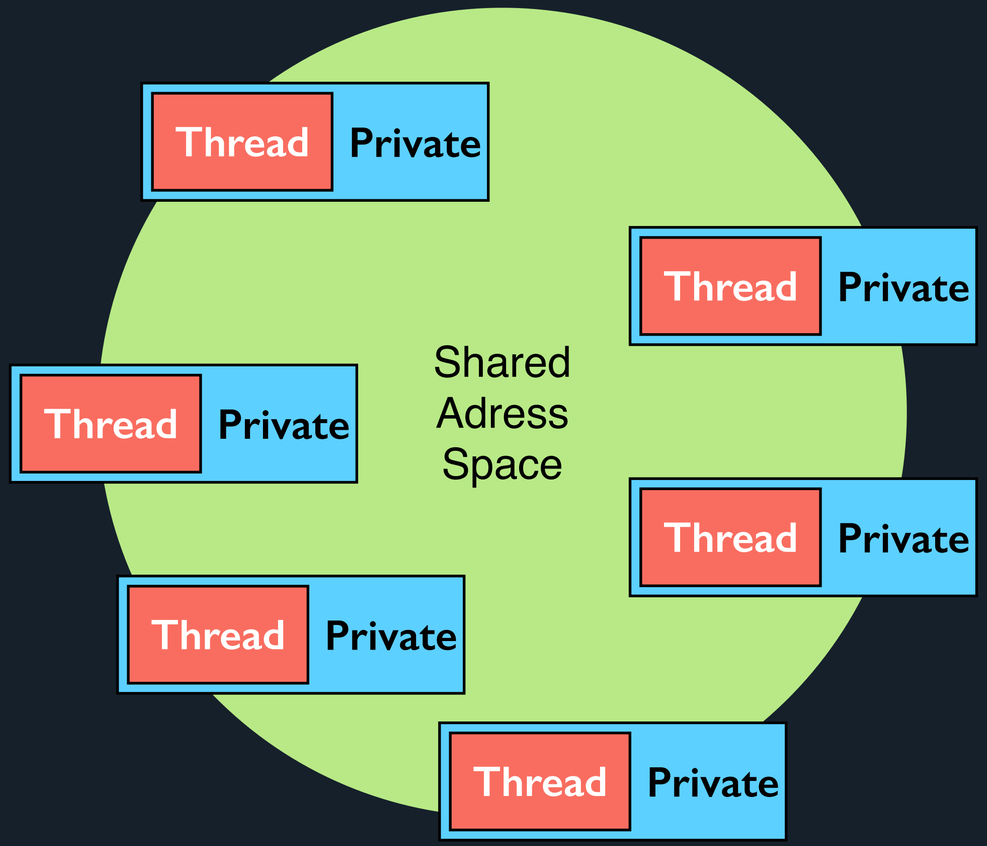
Sample output

```
[nshukla1@r033c01s04 RaceCondition]$ ./output
-----------------------------------------
 The total sum:          4371        4950
       runtime:        0.020324000
-----------------------------------------
[nshukla1@r033c01s04 RaceCondition]$ ./output
-----------------------------------------
 The total sum:          3570        4950
       runtime:        0.004176000
-----------------------------------------
[nshukla1@r033c01s04 RaceCondition]$ ./output
-----------------------------------------
 The total sum:          4824        4950
       runtime:        0.006123000
-----------------------------------------
```

SMP exhibits data race condition

An instance of a program:

- One Process and lots of threads
- Threads interact through reads/write to a shared address space
- OS scheduler decides when to run which threads .. interleaved for the fairness



Race condition:

when two or more threads access the same memory location

Synchronisation to assure every legal order results in correct results

Synchronization is used to impose order constraints

There are two fundamental approaches resolving race conditions:

Mutual Exclusion (Mutex)

Define a block of code that only one thread at a time can execute

Reduction

Synchronization is used to impose order constraints

protects accessing to shared data

High level synchronization

- Critical
- Atomic
- Barrier
- Order

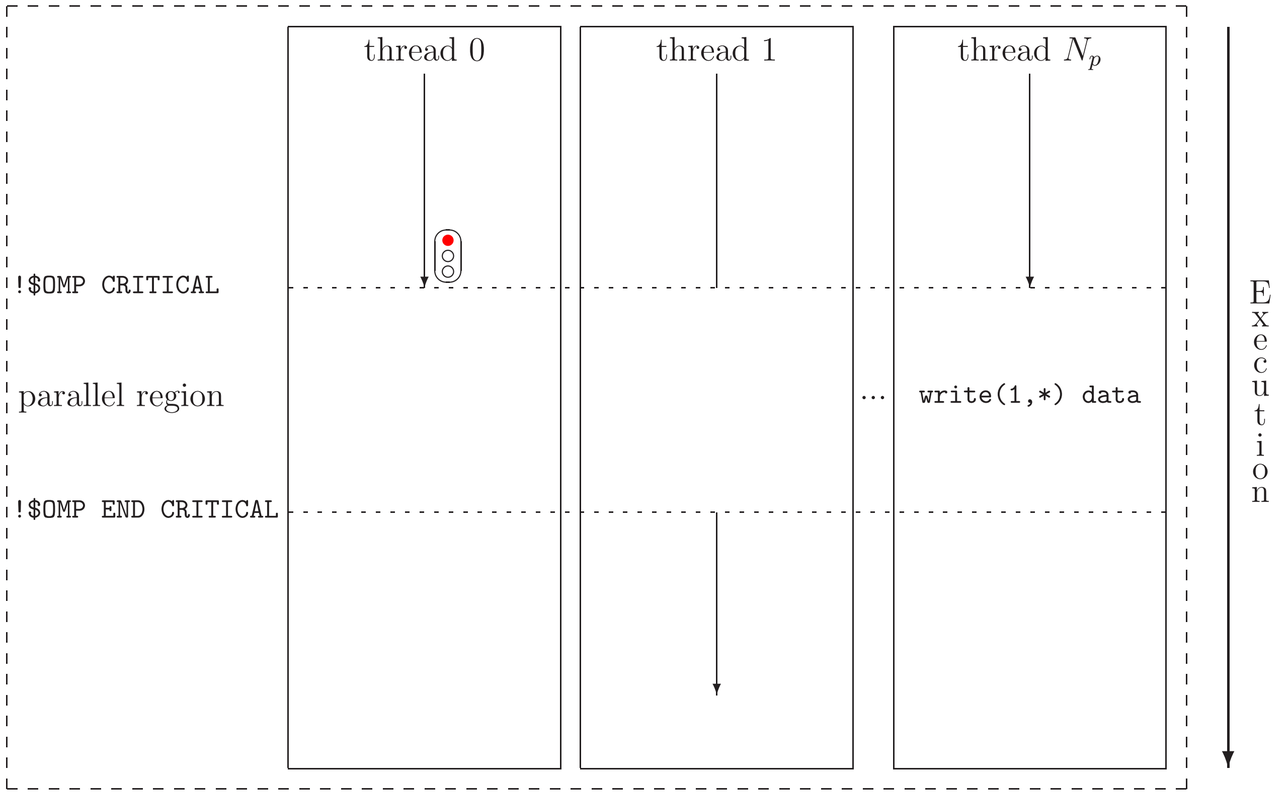
Low level synchronization

- Flush
- Locks (both simple and nested)

Critical section prevents multiple threads from accessing a section of code at the same time

Syntax of the critical construct in C/C++ and Fortran

#pragma omp critical *[(name)] new-line* *structured block*
!$omp critical *[(name)]* *structured block* **!$omp end critical** *[(name)]*



Avoiding race condition with CRITICAL construct

```
program race_condition
 USE omp_lib
 IMPLICIT NONE
 real(kind=8) :: start, finish
 integer i, n, sum
 n = 10000
 sum = 0
 Call cpu_time(start)
 !omp parallel do
 DO I = 1, n-1
  !omp critical
   sum = sum + I
  !omp end critical
 END DO
 !omp end parallel do
 Call cpu_time(finish)
 ! Print result --
  write(*, "The total sum : ", sum, ((n-1)*n)/2
  write(*, "(A, F18.9)") "RUNTIME : ", finish-start
end program race_condition
```

Sample output

```
[nshukla1@r033c01s04 RaceCondition]$ ./output
-----------------------------------------
 The total sum:    704982704   704982704
       runtime:        0.619179000
-----------------------------------------
```

Avoiding race condition with ATOMIC construct

```
program race_condition
 USE omp_lib
 IMPLICIT NONE
 real(kind=8) :: start, finish
 integer i, n, sum
 n = 10000
 sum = 0
 Call cpu_time(start)
 !omp parallel do
 DO I = 1, n-1
  !omp atomic
   sum = sum + I
  !omp end atomic
 END DO
 !omp end parallel do
 Call cpu_time(finish)
 ! Print result ––
  write(*, “The total sum : ”, sum, ((n-1)*n)/2
  write(*, “(A, F18.9)”) “RUNTIME : ”, finish-start
end program race_condition
```

Atomic constructs

lightweight mutex in OpenMP

guarantee mutually exclusive access to a specific memory location, represented through a variable

Sample output

```
[nshukla1@r033c01s04 RaceCondition_atomic]$ ./output
-----------------------------------------
 The total sum:    704982704   704982704
       runtime:        0.113553000
-----------------------------------------
```

Limitation of Atomic contracts

Read : operation in the form v = x

Write : operation in the form x = v

Update : operation in the form x++, x- -, - - x, binop= expr e.etc

Capture : operation in the form v = x++, v = x- etc

Where x and v are scalar variables

Binop is one of +, *, - etc

No “trickery” is allowed for atomic operations

- No operator overload
- No non-scalar types
- No complex expressions

Synchronization: Barrier

An instance of a program:

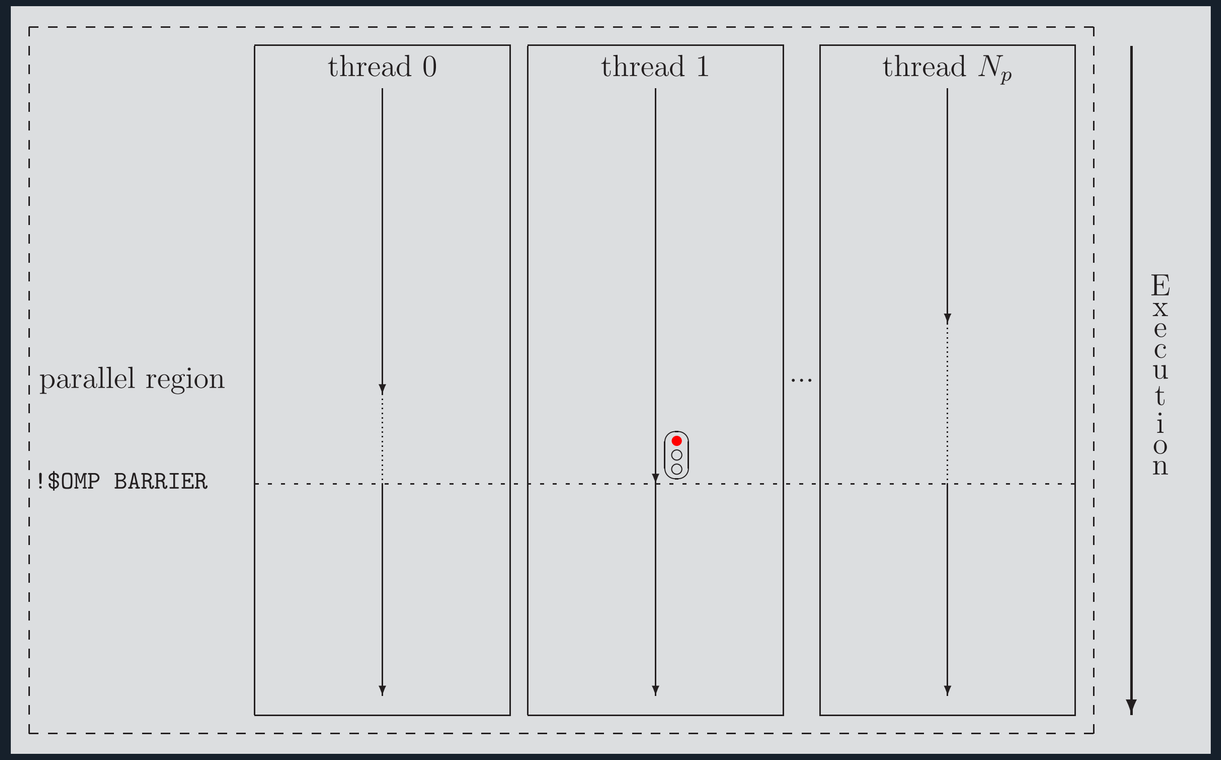
- This directive represents an explicit synchronization between the different threads in the team.
- No thread is allowed to continue until threads in a team reach the barrier

Advantage:

- Avoid data races to ensure the correctness of the program

Disadvantage:

- When a thread wait for other threads, it does not do anything useful and spends valuable resources



Synchronization: Barrier

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

int
main()
{
#pragma omp parallel
{
        printf("Hello from thread %d of %d\n", omp_get_thread_num(),
            omp_get_num_threads());
#pragma omp barrier // all threads wait here
        printf("Thread %d of %d have passed the barrier\n",
            omp_get_thread_num(), omp_get_num_threads());
}
        return 0;
}
```

```
$ ./std.out
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 0 of 4
Hello from thread 2 of 4
Thread 1 of 4 have passed the barrier
Thread 3 of 4 have passed the barrier
Thread 0 of 4 have passed the barrier
Thread 2 of 4 have passed the barrier
```

Explicit way

- #pragma omp barrier

Implicit way

- Parallel
- Loop
- Single

Try removing the barrier

Synchronization is used to impose order constraints

There are two fundamental approaches resolving race conditions:

Mutual Exclusion (Mutex)

Define a block of code that only one thread at a time can execute

Reduction

Reduction

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

int main()
{
  const int n = 1000;
  int total = 0;
  int I;
  printf ( "  ==================================\n" );
  printf ( "  no. processors = %d\n", omp_get_num_procs ( ) );
  printf ( "  no. of threads = %d\n", omp_get_max_threads ( ) );
  printf ( "  ==================================\n" );

  #pragma omp parallel for
   for (i = 0; i < n; i++)
         total = total + i;

  printf("  Total=%d (must be %d)\n", total, ((n-1)*n)/2);

  /*Terminate.*/
  printf ( "  Normal end of execution.\n" );
}
```

- Parallel tasks often produce some quantity that needs to be summed or otherwise combined.

How do we handle this case?

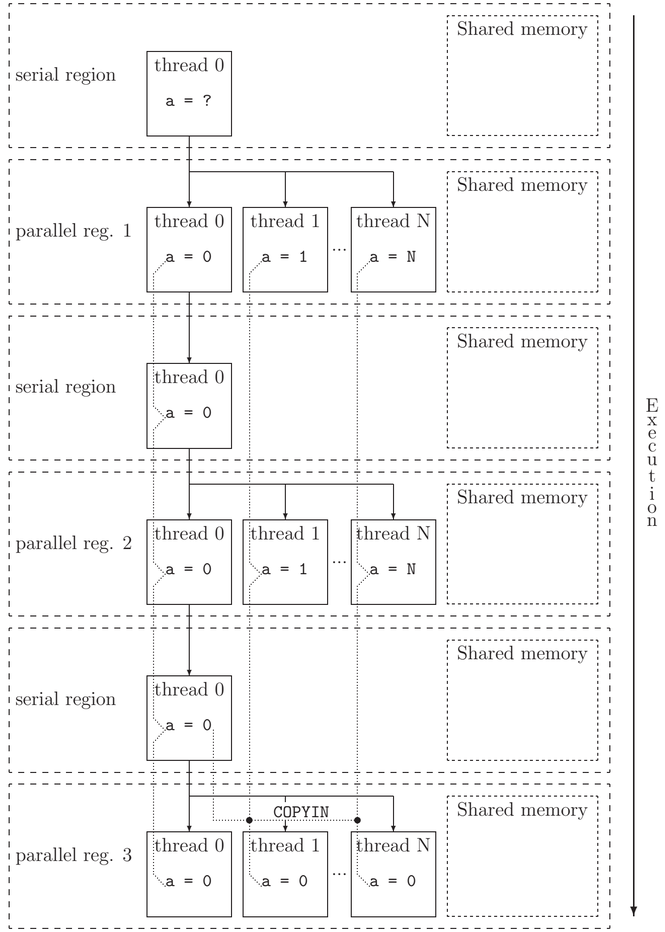
It is applicable to single line of code that performs read write update and capture

Reductions

- We are combining values into a single accumulation variable (sum)
- There is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation ... it is called a "reduction"

- `reduction(+:var)`
- `reduction(-:var)`
- `reduction(*:var)`
- `reduction(.and.:var)`
- `reduction(.or.:var)`
- `reduction(.eqv.:var)`
- `reduction(.neqv.:var)`
- `reduction(.max.:var)`
- `reduction(.min.:var)`
- `reduction(.iand.:var)`
- `reduction(.ior.:var)`
- `reduction(.ieor.:var)`

Much simpler to use the OpenMP reductions clause on a worksharing loop.



Reduction

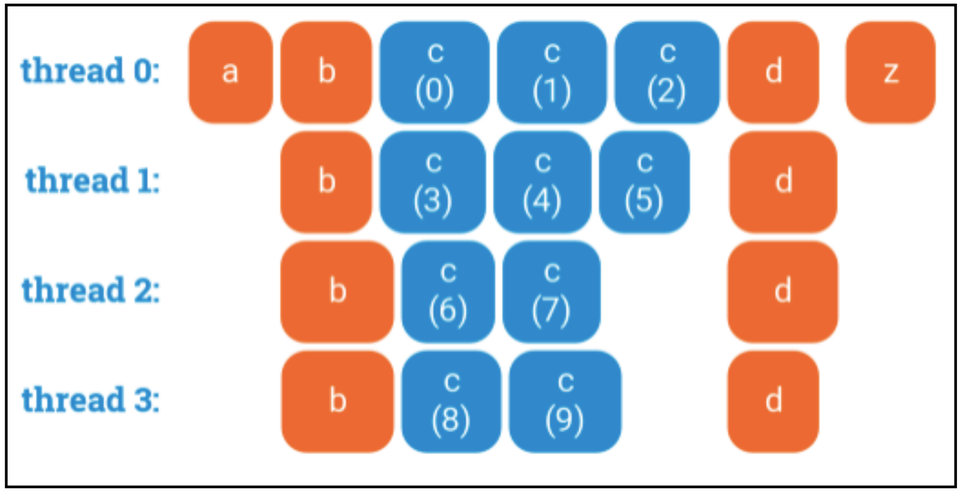
```
#include <stdio.h>
#include <omp.h>

int main()
{
  const int n = 1000;
  int total = 0;
  int I;
  printf ( "  ==================================\n" );
  printf ( "  no. processors = %d\n", omp_get_num_procs ( ) );
  printf ( "  no. of threads = %d\n", omp_get_max_threads ( ) );
  printf ( "  ==================================\n" );

  #pragma omp parallel for reduction(+:total)
    for (i = 0; i < n; i++)
         total = total + i;

  printf("  Total=%d (must be %d)\n", total, ((n-1)*n)/2);

  /*Terminate.*/
  printf ( "  Normal end of execution.\n" );
}
```

- An operation that "combines" multiple elements to form a single result, such as a summation, is called a reduction operation
- Syntax: reduction (operator:variable list)
- A private copy of each variable which appears in reduction is created as if the private clause is specified
- After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction

OpenMP parallel for loops: waiting

When you use a parallel region, OpenMP will automatically wait for all threads to finish before execution continues.
There is also a synchronization point **after** each omp for loop; here no thread will execute d() until all threads are done with the loop:

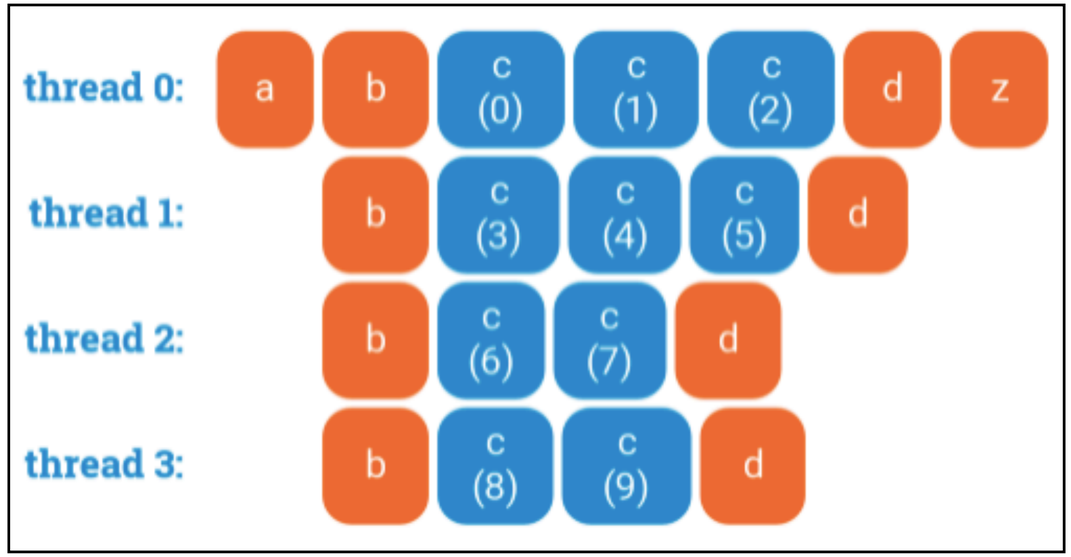
```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



Synchronization after the loop can be disabled using nowait

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

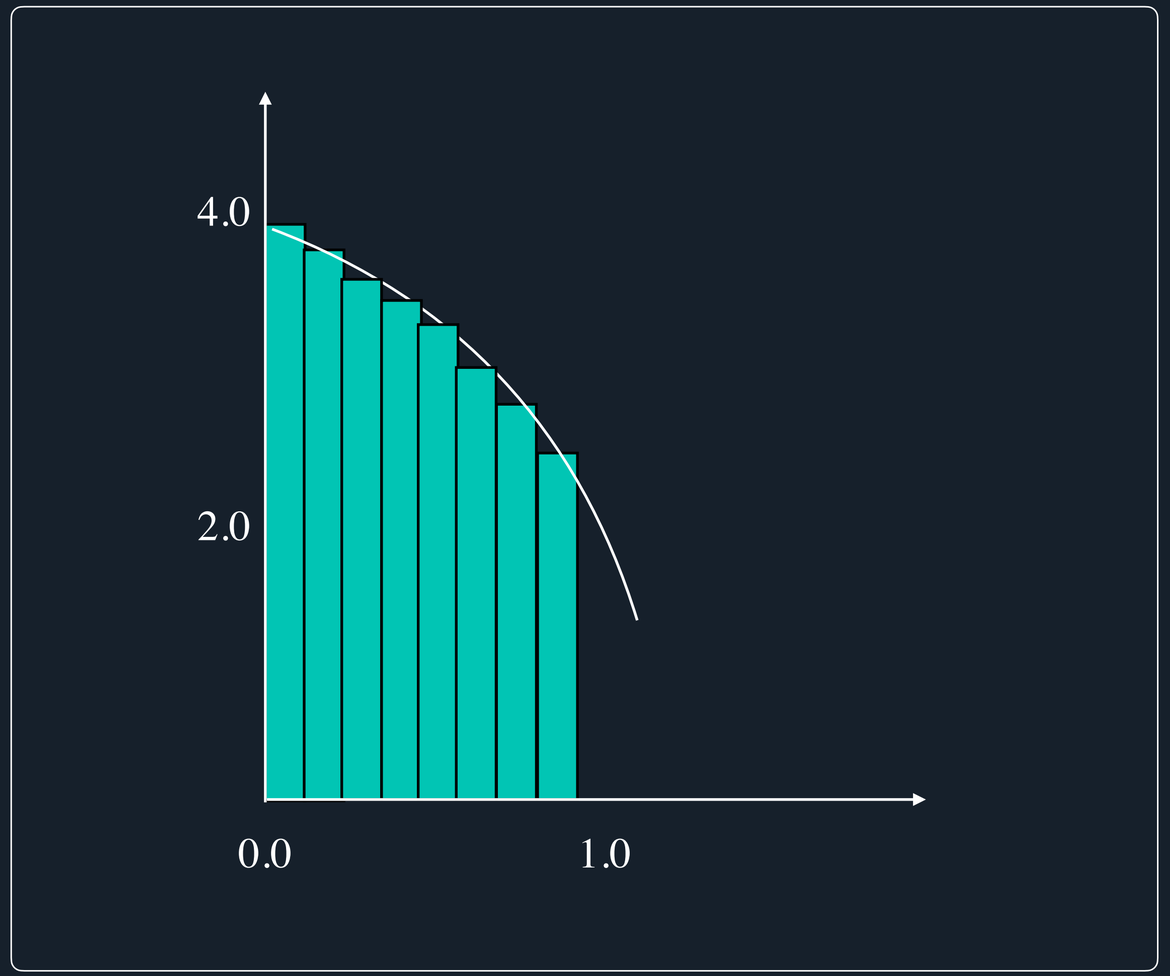
```
a();
#pragma omp parallel
{
    b();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    #pragma omp critical
    {
        d();
    }
}
z();
```



If you need a critical section after a loop, note that normally OpenMP will first wait for all threads to finish their loop iterations before letting any of the threads to enter a critical section:

Fourth exercise

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$



a) Use Critical and Atomic construct or reduction construct
b) Compile
c) Run and time it
d) Optional experiments with OMP_NUM_THREADS

Fourth exercise

$$\sum_{i=0}^{N} F\left(x_i\right) \Delta x \approx \pi$$

pi is: 3.1415926535904264
error is: 0.0000000000006333
runtime: 0.230

```
real(kind=8), parameter :: PI_8 = 4.0_8 * atan(1.0_8)

! step size is dependent upon the number of steps
dx = 1.0_8/num_steps

! Start timer
call cpu_time(start)

! main loop
do ii = 1, num_steps
  x = (ii-0.5_8)*dx
  sum = sum + 4.0_8/(1.0_8+x*x)
end do

pi = dx * sum

! Stop timer
call cpu_time(finish)
```

Welcome back

Using a critical to remove impact of false sharing

- x needs to be used independently by each thread, so mark as private
- sum needs to be updated by all threads, so leave as shared.
- All threads can update this value directly!
- A critical region only allows one thread to execute at any one time. No guarantees of ordering.

```
! main loop
!$omp parallel do private(x)
do ii = 1, num_steps
  x = (ii-0.5_8)*dx
  !$omp critical
  sum = sum + 4.0_8/(1.0_8+x*x)
  !$omp end critical
end do
!$omp end parallel do
```

```
pi is:      3.1415926535904317
error is:   0.0000000000006386
runtime:    111.341
```

Using an atomic to remove impact of false sharing

- A **atomic** region protects a whole block of code
- For a single operation, can use atomic instead.
- Atomic provides mutual exclusion but only applies to the update of a memory location

```
! main loop
!$omp parallel do private(x)
do ii = 1, num_steps
  x = (ii-0.5_8)*dx
  !$omp atomic
  sum = sum + 4.0_8/(1.0_8+x*x)
end do
!$omp end parallel do

pi = dx * sum
```

```
pi is:        3.1415926535901697
error is:     0.0000000000003766
runtime:      6.530
```

Reduction: Pi

- A private copy of each variable which appears in reduction is created as if the private clause is specified
- After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction

```
! Main loop
 !$omp parallel do private(x) reduction(+:sum)
 do ii = 1, num_steps
    x = (ii-0.5_8)*step
    x2 = 4.0_8/(1.0_8+x*x)
    sum = sum + x2
 end do
 !$omp end parallel do

 pi = step * sum
```

```
pi is:        3.1415926535897900
error is:     0.0000000000000031
runtime:      0.022
```

Runtimes

Implementation	Runtime(s)
Serial	0.230
Critical	111.341
Atomic	6.5
Reduction	0.022

All about OpenMP

OpenMP is a multi-threading, shared address model

- Threads communicate by sharing variables
- By default, all the data is available to all threads
- There is a single copy of shared data

Sharing memory

- Creating a parallel region and distributing work among threads

Data scoping

- How is data shared and made visible across threads
- PRIVATE, SHARED, DEFAULT, FIRSTPRIVATE

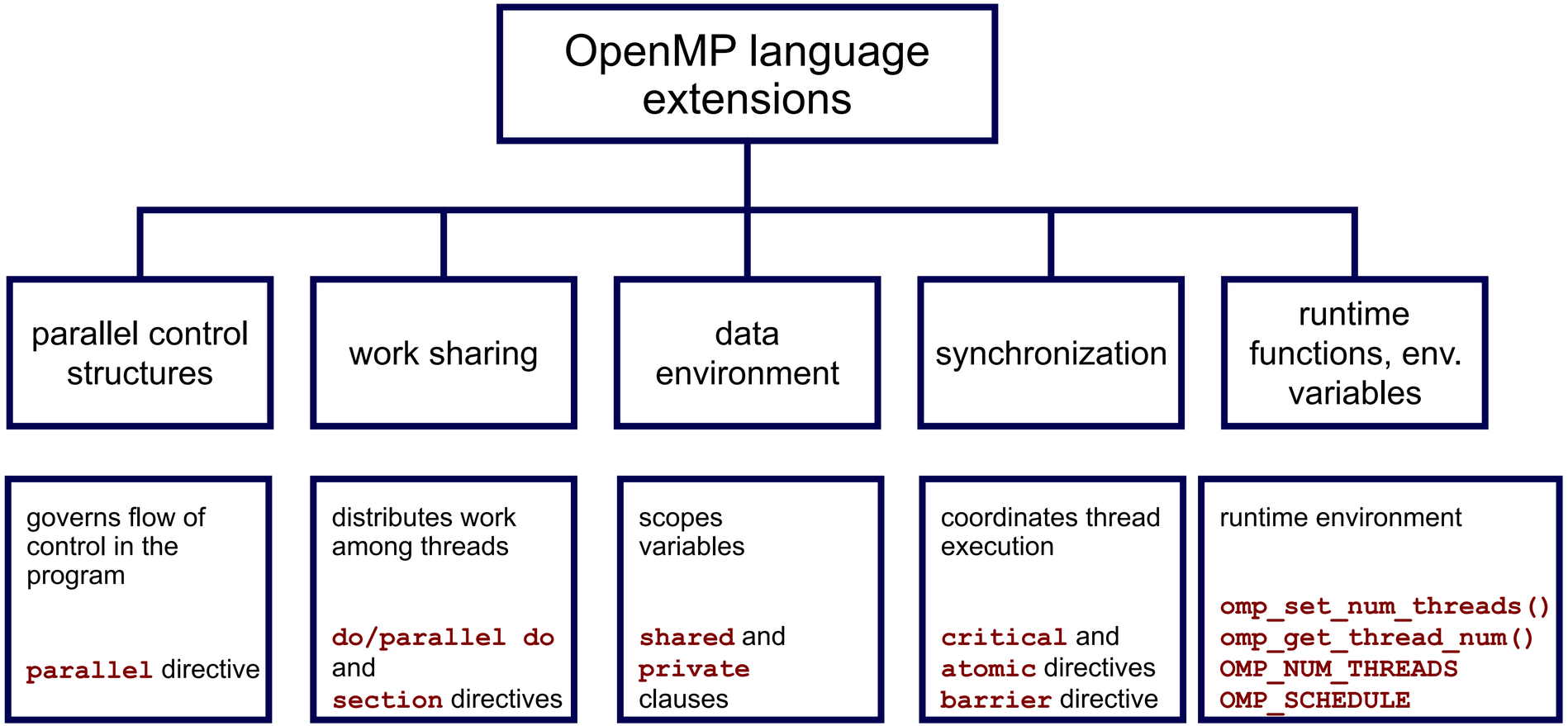
Sharing of data causes race conditions

- Race condition: when the program's outcome changes as the threads are scheduled differently
- Race condition needs to be managed

Synchroinzation is expensive

- Change how data is accessed to minimise the need for synchronization
- Synchronization to protect data conflicts

Closing thoughts



Most OpenMP programs only use these 21 items

pragma, env variable, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
#pragma omp barrier/critical	Synchronization and race conditions, interleaved execution.
#pragma omp for/parallel for	Worksharing, parallel loops, loop carried dependencies.
#pragma omp single	Workshare with a single thread.
#pragma omp task/ taskwait	Tasks including the data environment for tasks.
setenv OMP_NUM_THREADS N	Setting the internal control variable (ICV) for the default number of threads with an environment variable
void omp_set_num_threads() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and ICV. SPMD pattern: Create threads in a parallel region and split up the work.
double omp_get_wtime()	Speedup and Amdahl's law, false sharing and other perf issues.
reduction(op:list)	Reductions of values across a team of threads.
schedule (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(none)	Force explicit definition of each variable’s storage attribute
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive).

Grazie Mille!!

Feel free to reach me out
n.shukla@cineca.it