OpenMP

http://openmp.org/wp/

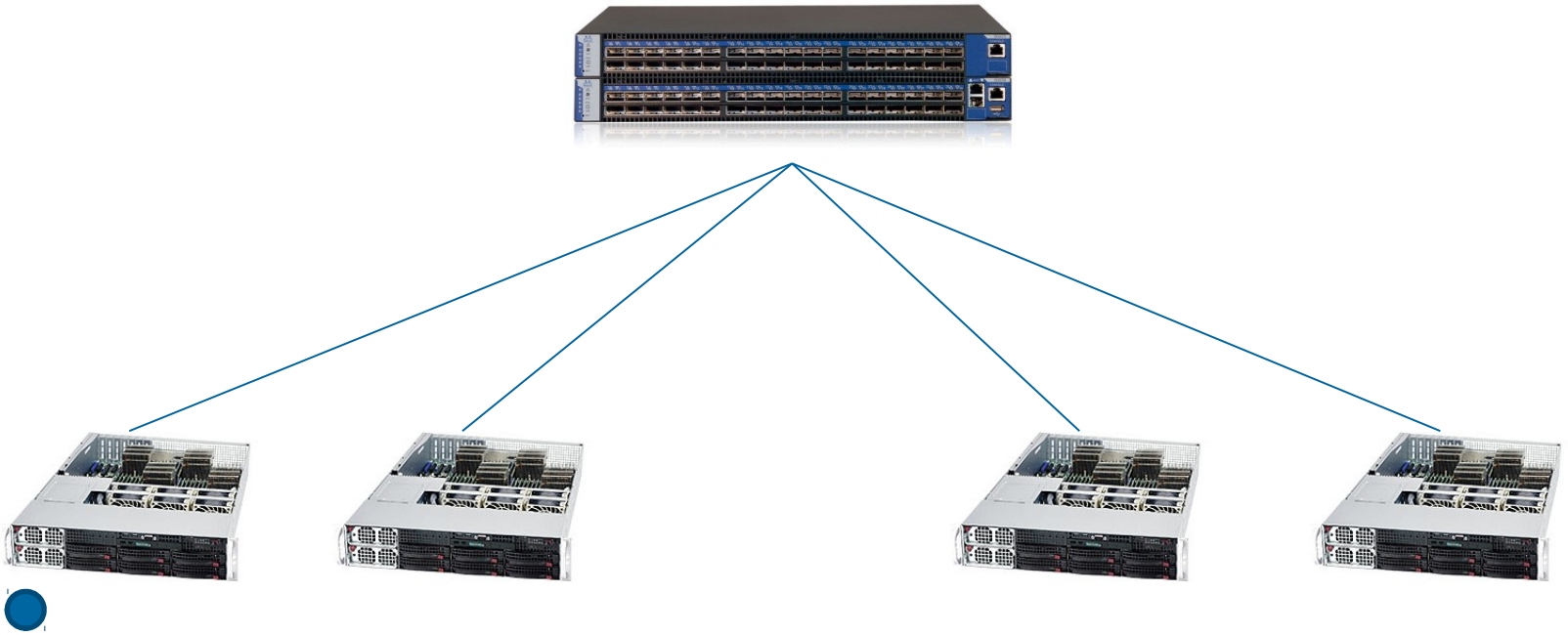
17-10-2016

INTRODUCTION

17-10-2016

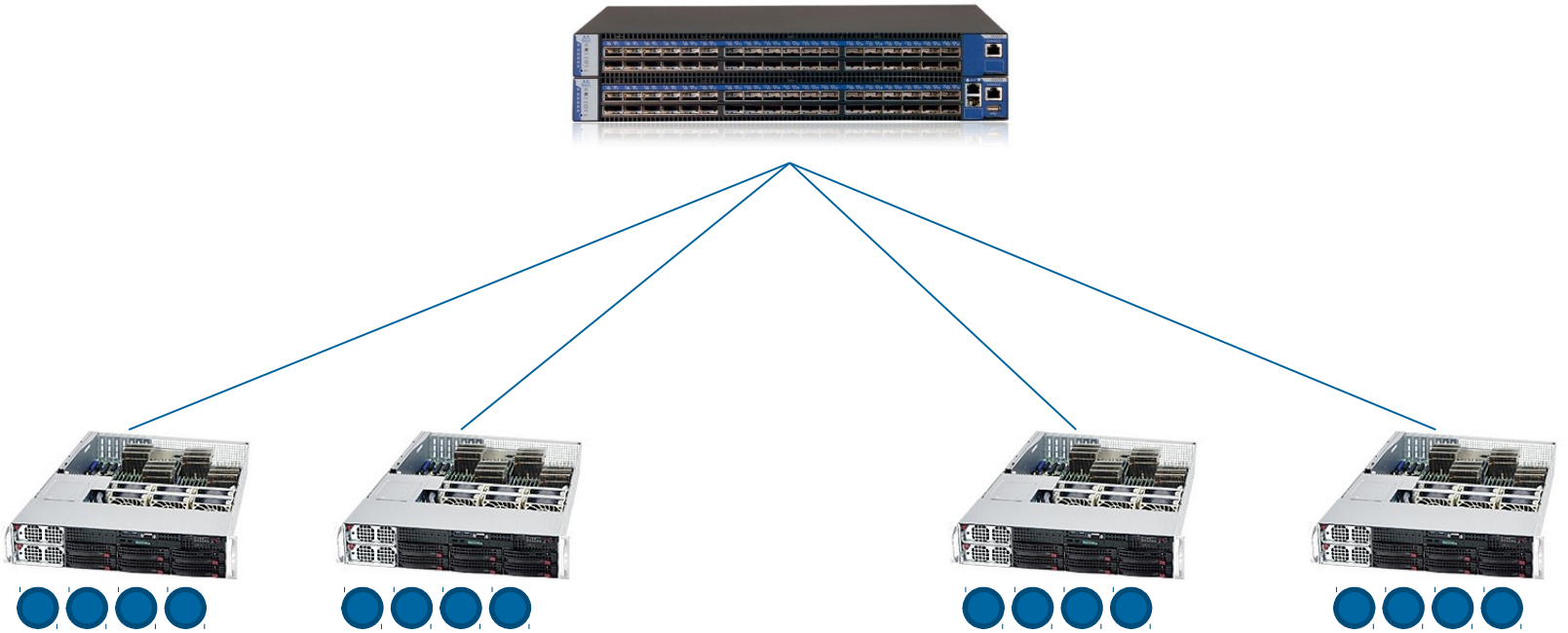
Why OpenMP ?

- cluster is composed by several compute nodes. Your sequential application will run **within a node**.



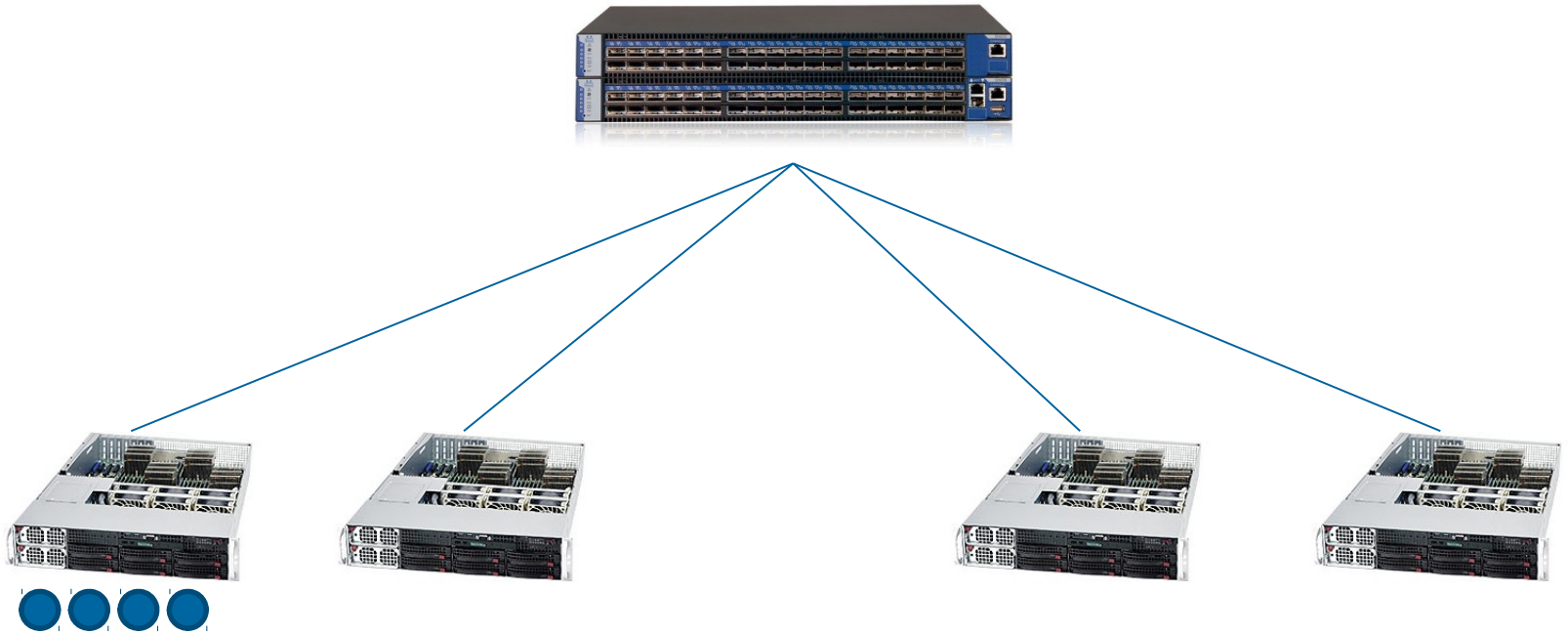
Why OpenMP ?

- MPI is a library which will be used to run your application in parallel on all nodes and using **all cores of each node.**



Why OpenMP ?

- OpenMP allows to run your application in parallel **within a node** but using **all cores.**



What is OpenMP?

- A standard API for parallel shared memory application
- For C/C++ or Fortran
- Composed of compilation directives, a library and a set of environment variables
- Allows to create and manage threads:
  - based on the ‘fork and join’ model
- An OpenMP program is portable
- Easy to program

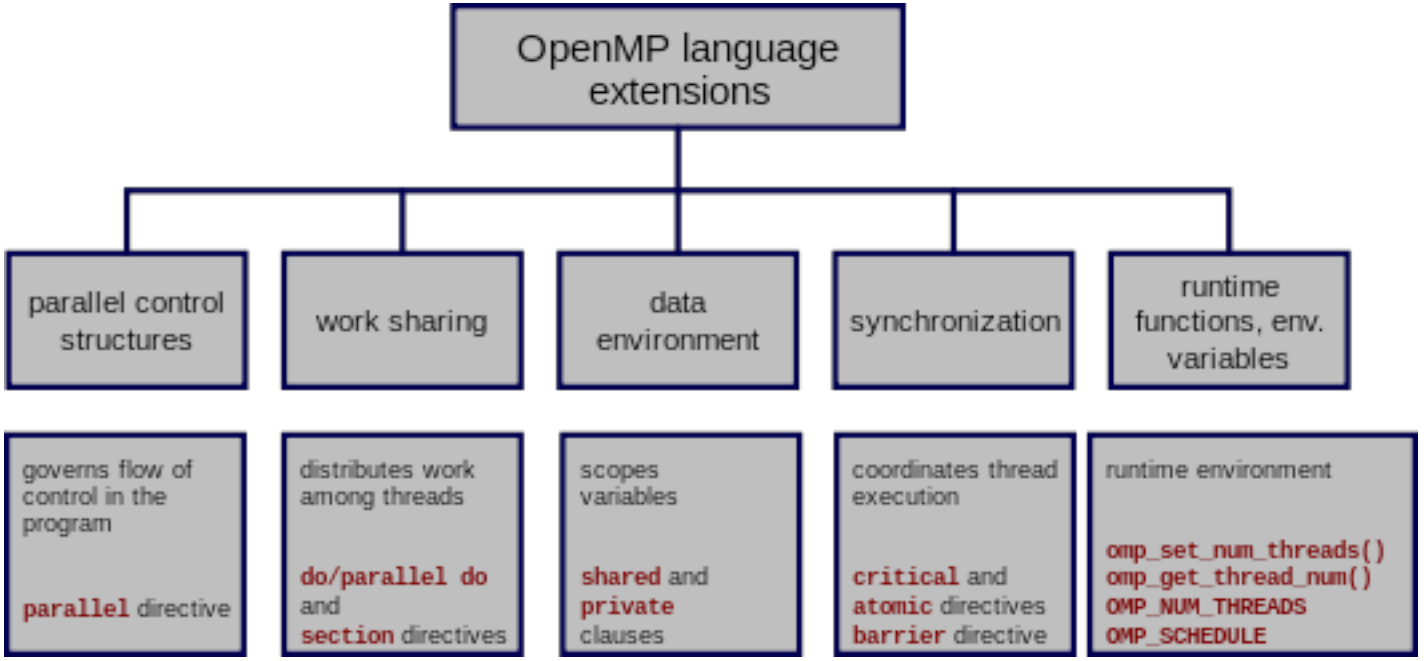
Processes and Threads (1/2)

- A process executes an instance of a program
- Each process has its own address space
  - processes are independent from each other
  - the synchronization between processes is managed at the kernel level
- The creation of a process is ‘heavy’
- A process can contain several threads
- A thread is an entity inside a process

Processes and Threads (2/2)

- Threads can be seen as lighter processes
  - they are similar to processes
    - they have their own control flow
  - but their creation cost is cheaper:
    - no need to create a separate address space
    - they share the address space of the process in which they are
    - they also share the context of this process
- The synchronization between threads is cheaper as it can be done at the process level
  - can be done via a variable in the common address space
- Changing the context between two threads is faster than between two processes

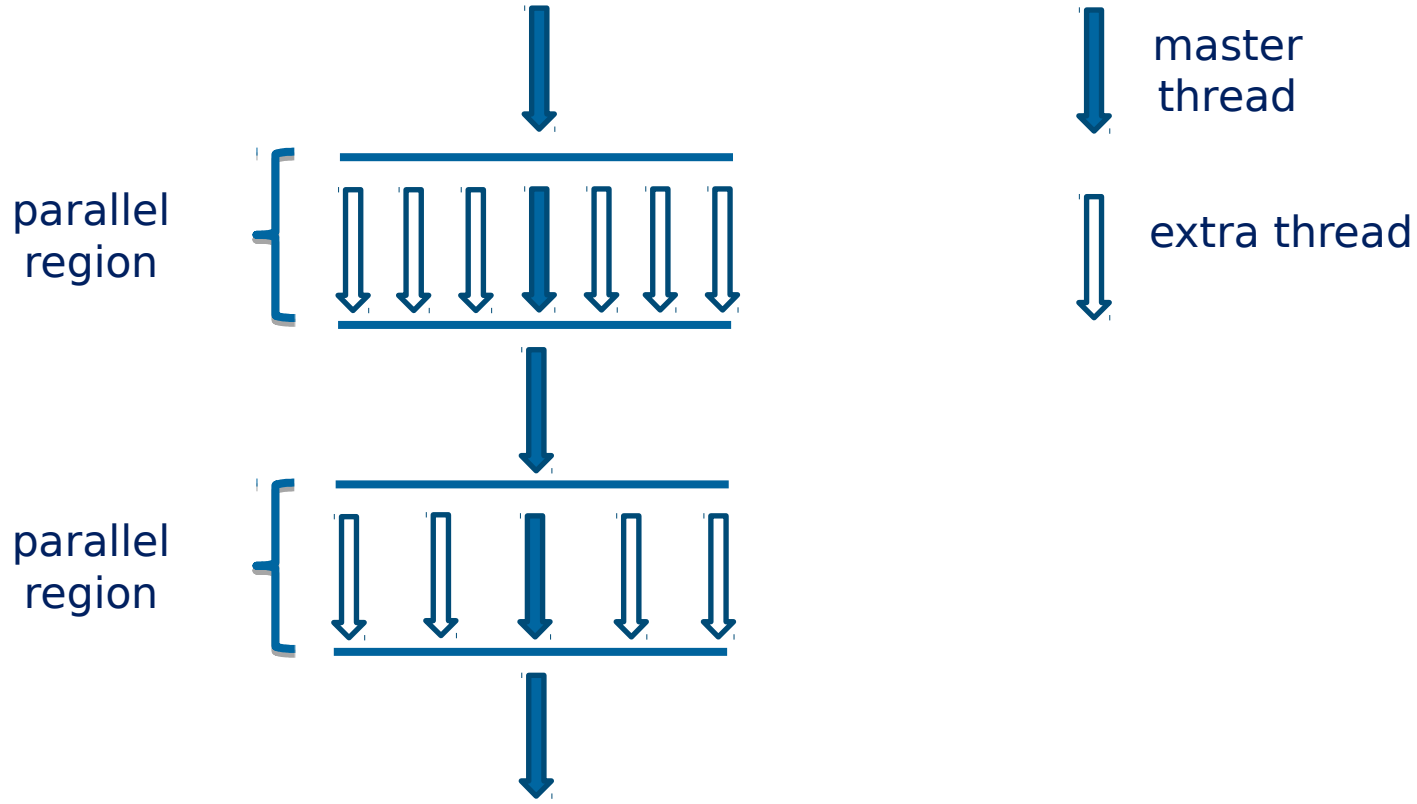
OpenMP Toolbox



The ‘Fork and Join’ Model

- OpenMP is based on the ‘fork and join’ model
- At the beginning of the execution, only the ‘master thread’ is active
- The ‘master thread’ executes the sequential parts of the application
- When entering a parallel region of the program, the master thread creates or wakes up additional threads
- When leaving a parallel region, the master thread destroys or suspends the additional threads

The ‘Fork and Join’ Model



OpenMP Directive Format

- C/C++

```
#pragma omp DirectiveName [clause[ [,] clause]...] \
#pragma omp                [clause[ [,] clause]...]
```

- ‘free form’ Fortran

```
!$omp nomDirective [clause[ [,] clause]...] &
!$omp              [clause[ [,] clause]...]
```

- ‘fixed form’ Fortran

```
directive DirectiveName [clause[ [,] clause]...] newligne

with directives : !$omp ou c$omp ou *$omp
```

Examples of Library Routines

- Get the current thread identifier (ID)

```
int omp_get_thread_num(void);
```

- Get the number of threads in the current parallel region

```
int omp_get_num_threads(void);
```

- Set the number of threads in the current parallel region

```
void omp_set_num_threads(int);
```

Examples of Environment Variables

- Set the number of threads used in parallel regions

```
OMP_NUM_THREADS
```

- Set the maximum number of threads available for the OpenMP program

```
OMP_THREAD_LIMIT
```

Conditional Compilation

- OpenMP directives are
  - translated by the compiler if a proper command line option is given to the compiler
  - considered as comments otherwise

- This allows to have an unique code for a sequential or parallel execution
  - it is simply annotated with directives

ifort **–qopenmp** –O3 …. foo.f90

OpenMP Memory Model

- The OpenMP API provides a relaxed-consistency, shared-memory model
- All OpenMP threads have access to a place to store and to retrieve variables, called the *memory*
- Each thread is allowed to have its own *temporary view* of the memory
- This temporary view avoids a thread going to memory for every reference to a variable
  - allow the use of register, cache or other local storage
- Each thread also has access to another type of memory that must not be accessed by other threads, called *threadprivate memory*

Directives, Constructs and Regions

- OpenMP **constructs** are made with OpenMP **directives**
- **Constructs** define **regions**

directive construct region

```
#pragma omp my_directive
{
  ...
  foo();
  ...
}
```

```
foo()
{
  ...
  ...
}
```

An example

```
program parallel

implicit none
real :: a
a= 13450

print*, « A = » , a

end program parallel
```

```
> ifort                file.f90

> ./a.out
```

```
A = 13450
```

An example

```
program parallel
use OMP_LIB
implicit none
real :: a
a= 13450
!$OMP PARALLEL
print*, « A = » , a
!$OMP END PARALLEL
end program parallel
```

```
> ifort –openmp file.f90
> export OMP_NUM_THREADS=4
> ./a.out
```

```
A = 13450
A = 13450
A = 13450
A = 13450
```

Parallel Construct

17-10-2016

The ‘parallel’ Construct

- Is declared using the **parallel** directive
- Implements the ‘fork and join’ model
  - A thread that encounters a ‘parallel’ region
    - creates a **team** of threads
    - belongs to the team
    - is the **master** thread of the region
- Each thread of the team executes the code in the parallel region (duplication)
- There is an implied barrier (synchronization) at the end of a **parallel** region

The ‘parallel’ Construct

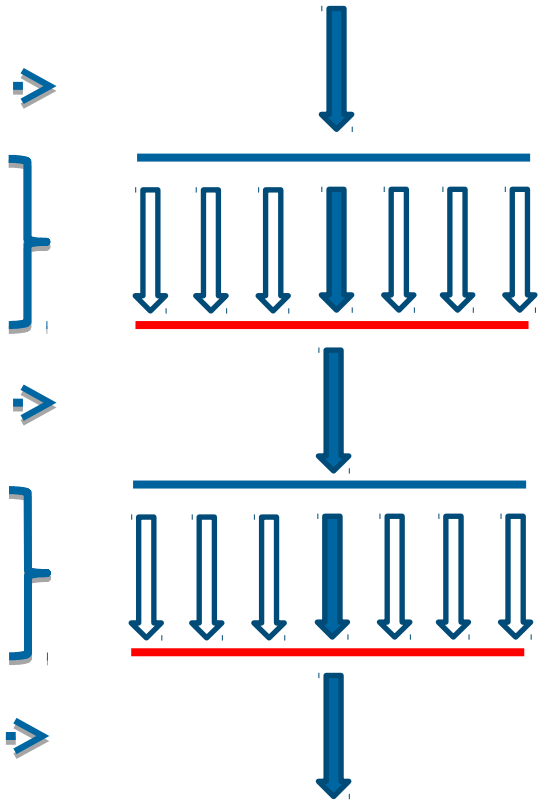
- The syntax of the **parallel** construct is as follows
  - C/C++

```
#pragma omp parallel [clause[ [,] clause]...]
{
  ...
}
```

```
!$omp parallel [clause[ [,] clause]...]
...
!$omp end parallel
```

Parallel Regions

```
//sequential part

#pragma omp parallel
{
  //parallel region
}

//sequential part

#pragma omp parallel
{
  //parallel region
}

//sequential part
```



master thread

extra thread

implied barrier

Atos

Scope of a Parallel Region

- The scope of a parallel region affects
  - the code lexically in the parallel construct
  - but also the code inside the sub-programs called from this construct

```
#pragma omp parallel
{
  int a = 4;
  int b = 5;

  myfunc(a , b);

}
```

```
void myfunc(int a, int b){

  printf(‘‘%d %d\n’’,a,b);

}
```

Parralel Region

- Each thread has an unique integer identifier
- The master thread of the region always has the 0 identifier
- **Branches into or out of the parallel region are not allowed**

```
#pragma omp parallel
{
   ...
  goto out;
}
out:
...
```

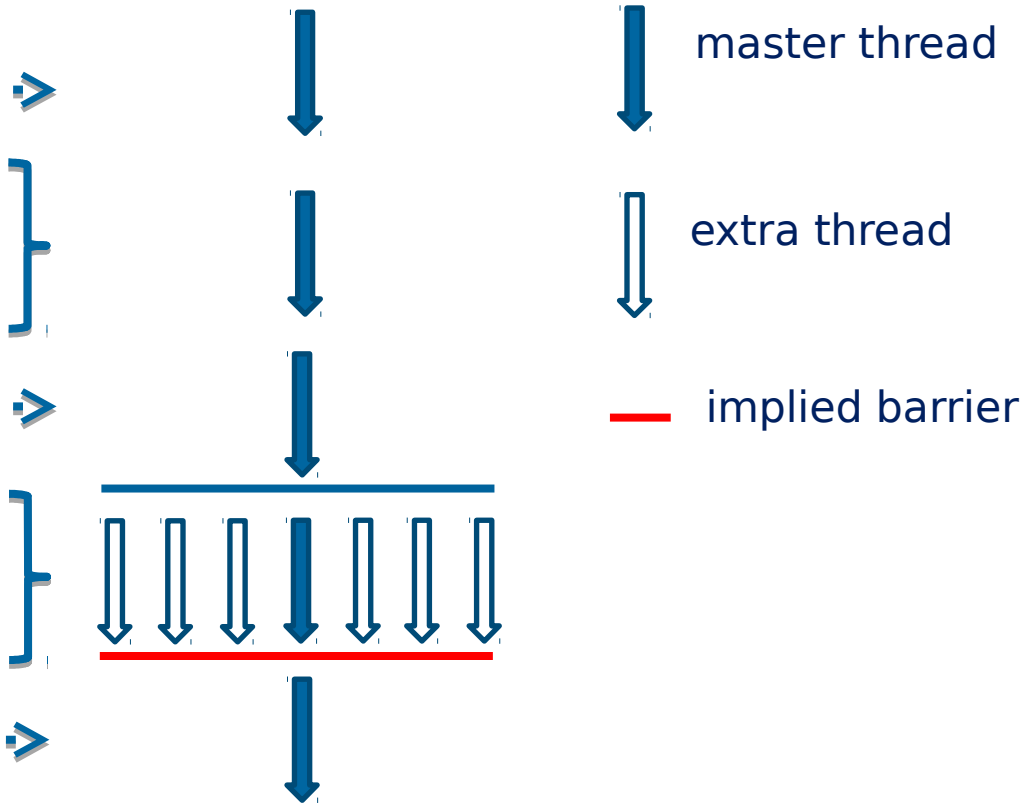
‘if’ and ‘num_threads’ Clauses

- The **if**(expr) clause determines the execution of a parallel region
  - if the *expr* expression evaluates to *false*, the region is inactive

- The **num_threads**(exp) clause specifies the number of requested threads for the parallel region
  - *exp* must be a positive integer

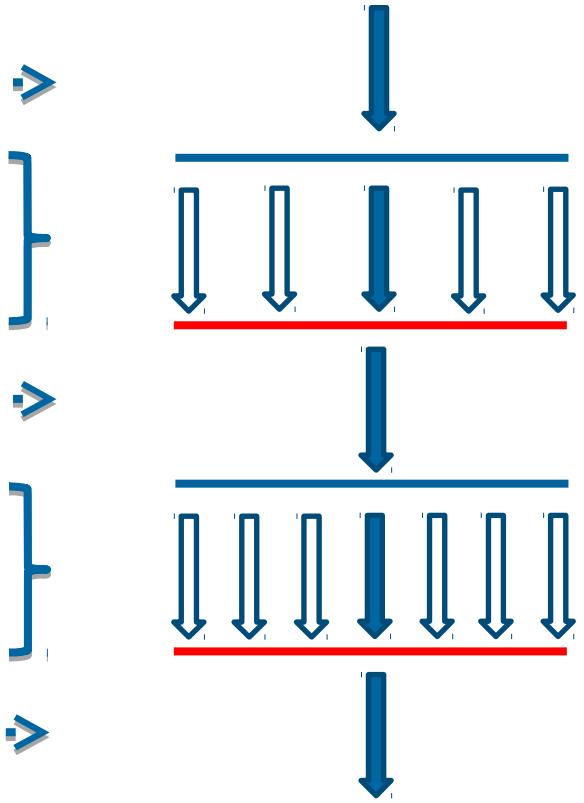
‘if’ Clause Example

```
//sequential part
int i = 6;

#pragma omp parallel if(i>9)
{
  //parallel region
}

//sequential part

#pragma omp parallel if(i>5)
{
  //parallel region
}

//sequential part
```



‘num_threads’ Clause Example

```
//sequential part

#pragma omp parallel num_thread(5)
{
  // parallel region
}

//sequential part

#pragma omp parallel num_thread(7)
{
  //parallel region
}

//sequential part
```



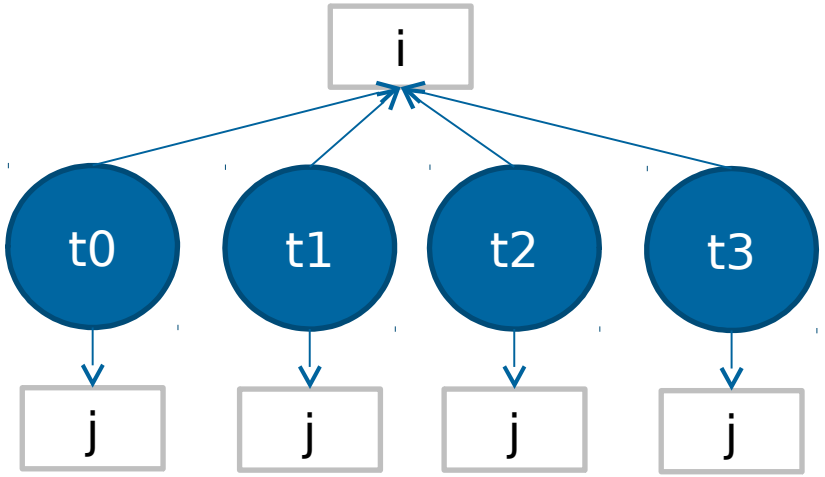
Data-Sharing Attribute

17-10-2016

Data-Sharing Attribute

- By default in parallel region
  - if a variable is declared inside the parallel region, each thread has a private copy of it
  - if a variable is declared outside the parallel region, all the threads share the same variable (global included)

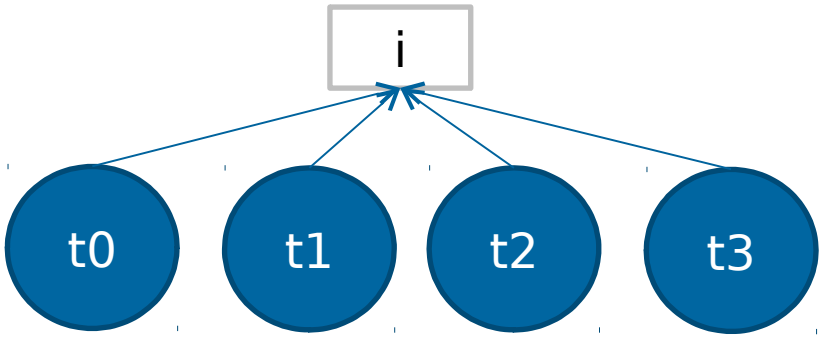
```
int i;

#pragma omp parallel
{
  int j;

  // 'i' is shared between all threads
  // each thread has its 'j'

}
```



The ‘shared’ Clause

- The **shared**(list) clause specifies a list of variable to be shared between threads

```
int i ;

#pragma omp parallel shared( i )
{
  //all threads share ‘i’
}
```

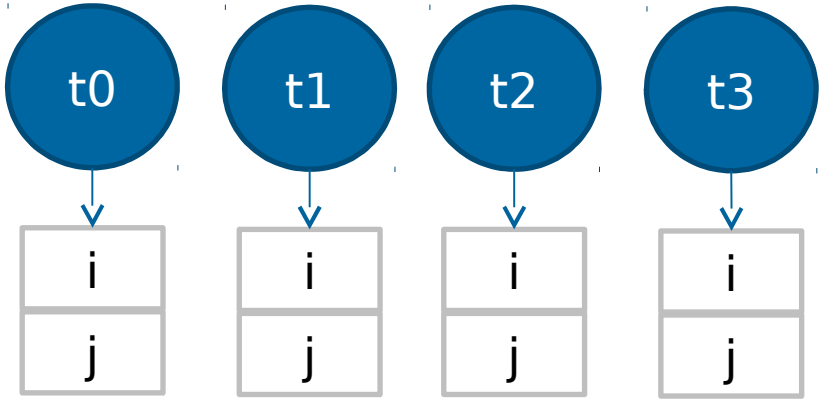


- It is the default attribute for variables declared outside a parallel region

The ‘private’ Clause

- The **private**(list) clause specifies a list of variable for which each thread will have their own local private copy

```
int i = 15;

#pragma omp parallel private( i )
{
  int j;

 //each thread has its  own ‘j’ and ‘i’

}
```

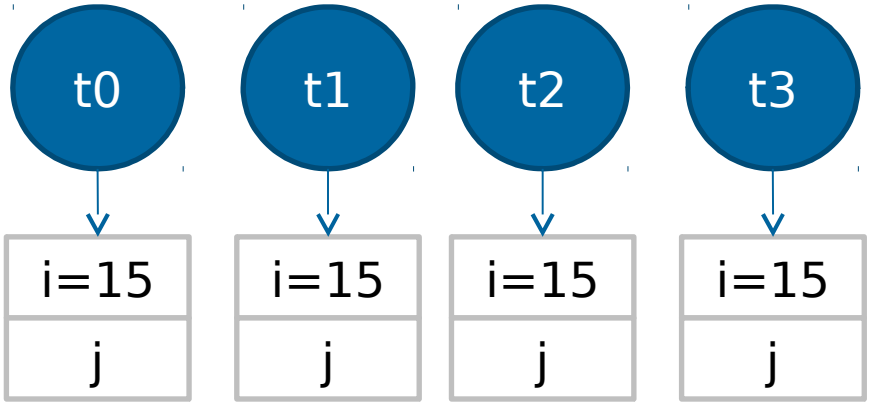


- Warning: the value of a variable declared private is undefined at the beginning of a parallel region
  - Do not expected ‘i’ to be 15 in the above example

The ‘firstprivate’ Clause

- The **firstprivate**(list) clause specifies a list of variable to be
  - private for each thread
  - initialized with the value of the variable before entering the parallel region

```
int i = 15;

#pragma omp parallel firstprivate(i)
{
  int j;

 //each thread has its  own ‘j’ and ‘i’
 // ‘i’ is initialized to 15

}
```



The ‘default’ Clause

- The **default** clause specifies the default data-sharing attribute of variables inside the parallel construct
- There are 4 possible default attribute in Fortran
  - default(**shared | none | private | firstprivate**)
- 2 in C/C++
  - default(**shared | none**)
- « none » forces the user to specify the data-sharing attribute for all variables

Worksharing for/do

17-10-2016

The ‘for/do’ Construct

- Apply to a loop or loopnest
- Share the work of the loop among the threads (worksharing)
- Tell the compiler that the loop is parallel
  - The user has to ensure that the loop is parallel
- There is an implied synchronization barrier between threads at the end of the loop
- The induction variable is automatically private
- Restrictions
  - The number of iterations must be able to be evaluated before the loop execution
  - The loop must not contain any break, return, goto or exit

Example of a ‘for’ Construction

```
int i;

#pragma omp parallel
{

   #pragma omp for
    for(i=0; i<N; i++)
    {
       //iterations are spread between the threads
       //the induction  variable « i » is automatically private
       …
    }// implied synchronization


 }// implied  synchronization
```

The ‘for/do’ Construct

- The ‘for/do’ construct is made using the ‘for/do’ directive
  - C/C++

```
#pragma omp for[clause[ [,] clause]...]
```

  - Fortran

```
!$omp do[clause[ [,] clause]...]
...
[!$omp end do [nowait] ]
```

- Possible clauses
  - private(list), firstprivate(list), lastprivate(list),
  - reduction(operator: list), schedule(kind[, chunk size]), collapse(n)
  - ordered, nowait

Advice for the ‘for/do’ Construction

- Find intensive computational loop or loop nest
  - Worksharing between threads implies some cost (threads creation/management)
  - To parallelize a loop can sometimes slow down its execution!

Clauses of the ‘for/do’ Directive

17-10-2016

The ‘lastprivate’ Clause

- The **lastprivate**(list) clause specifies
  - a shared variable is made private to each thread
  - that, after the ‘for/do’ region, the value of the variable is the value computed by the thread which executes the last iteration

```
int i,val;

#pragma omp parallel
 {
  #pragma omp for lastprivate( val )
  for(i=0; i<n; i++)
  {
     val=i;
  }

  //here ‘val’ is shared and its value is n-1
}
```

The ‘nowait’ Clause (1/2)

- The **nowait** clause allows the user to remove the implied barrier at the end of the ‘for/do’ region

- C/C++:

```
#pragma omp for nowait
{
  ...
}
```

- Fortran:

```
!$omp do
  ...
!$omp end do nowait
```

The ‘nowait’ Clause (2/2)

```
int i;

#pragma omp parallel
{

  #pragma omp for
   for(i=0; i<N; i++)
   {
    …
   }// implied synchronization barrier


   #pragma omp for nowait
   for(i=0; i<N; i++)
   {
    …
   }// no implied synchronization barrier


 }// implied synchronization barrier
```

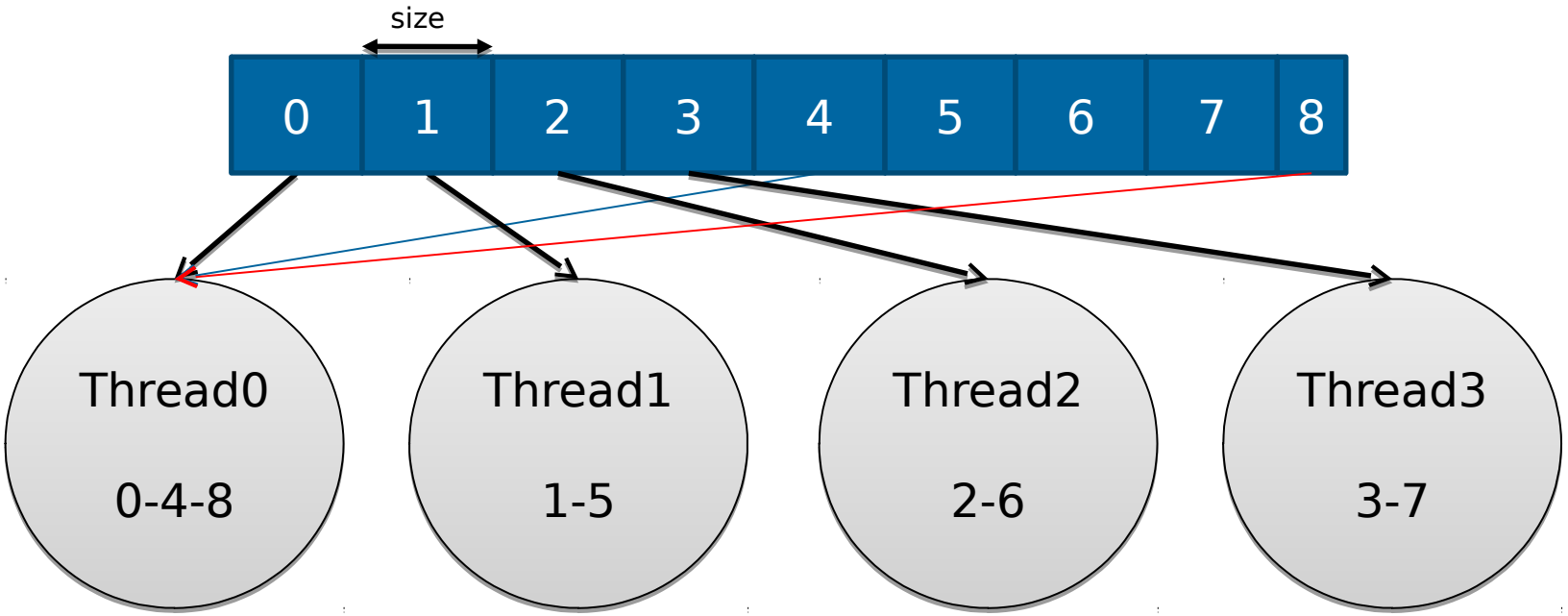
- Be careful of race conditions!!!

The ‘schedule’ Clause

- The schedule clause specifies the distribution of iterations among the threads
- There are 5 possible distribution modes
  - schedule(**static** [,chunk])
  - schedule(**dynamic** [,chunk])
  - schedule(**guided** [,chunk])
  - schedule(**auto**)
  - schedule(**runtime**)

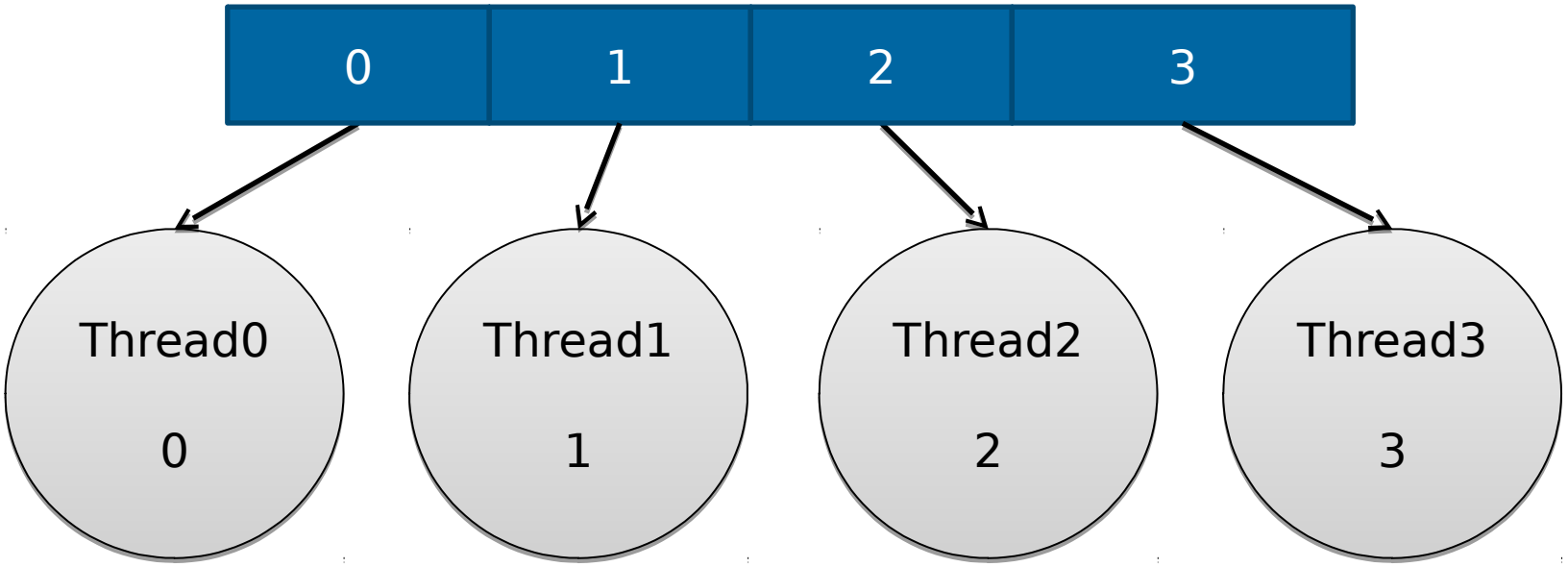
schedule(static[,*chunk_size*]) (1/3)

- The iteration space is split in chunks of size *chunk_size*
- The chunks are assigned to the threads in a round-robin fashion in the order of the thread number



schedule(static[,size]) (2/3)

- When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread
  - Note that the size of the chunks is unspecified in this case.



schedule(static[,chunk]) (3/3)

- The **schedule(static**[,*chunk_size*]**)** clause ensures the same iteration distribution between the threads of two consecutive loops if…
  - both loops have the same number of iterations
  - the chunk size is the same for both loops
  - both loops bind to the same parallel region

- In this case, data dependencies between the same iteration of two consecutive loop are guaranteed
  - The **nowait** clause can be used without risk of race condition

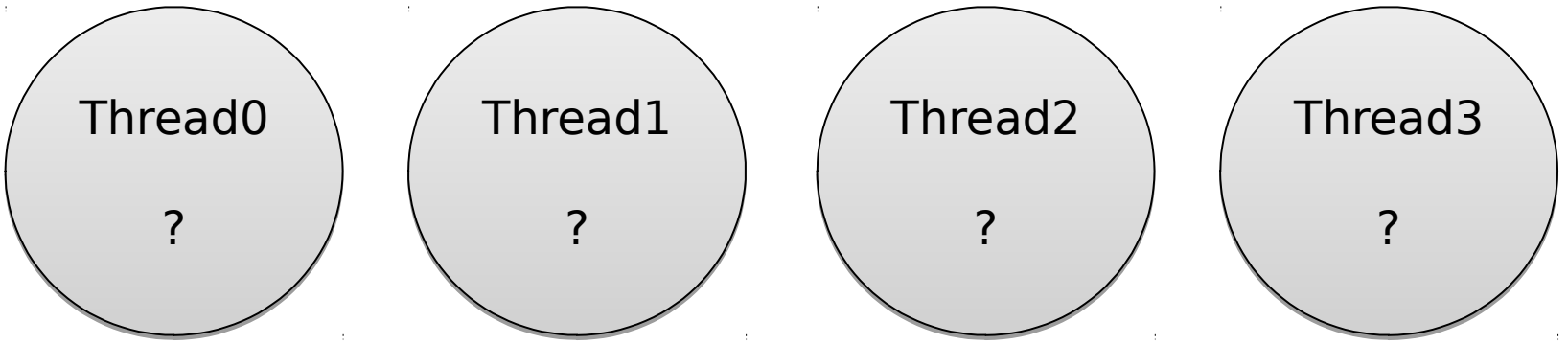
Execution Without Race Condition Example

```
int A[N];
for(i=0;i<N;i++)
    A[i] = 0;

#pragma omp parallel
{
 #pragma omp for schedule(static) nowait
 for(i=0;i<N;i++){
    A[i] = i;
 }

 #pragma omp for schedule(static)
 for(i=0;i<N;i++){
    A[i] += 1;
 }
}
```

schedule(dynamic[,*chunk_size*]) (1/2)

- The iteration space is distributed in chunks of size *chunk_size*
  - When no chunk size is specified, it defaults to 1

- When a thread finishes the execution of a chunk, it requests for a new chunk



schedule(guided[,*chunk_size*])

- The schedule(guided) clause divides the iteration space into smaller and smaller chunks
- The size of each chunk is proportional to the undistributed iterations divided by the thread number
- If a chunk size is specified, the minimum size of a chunk will be *chunk_size*
- The chunk distribution is the same as the one of the schedule(dynamic) clause; when a thread finishes a chunk execution, it requests a new chunk

schedule(guided[,*chunk_size*])

- Distribution example

```
#pragma parallel num_thread(4)
{
  #pragma omp for schedule(guided)
  for(i=0;i<256;i++)
  ...
}
```

  - a) there are 256 undistributed iterations
    - 1st chunk: 32 iterations= (256 / 4) / 2
  - b) 224 undistributed iterations remains
    - 2nd chunk : 28 iterations = (224 / 4) / 2
  - c) 196 undistributed iterations remains
    - 3rd chunk : 24 iterations = (196 / 4) / 2
  - ...

schedule(auto) and schedule(runtime) Clauses

- schedule(auto)
  - The compiler and/or the runtime takes the decision of the iteration space distribution
  - This distribution is implementation dependent

- schedule(runtime)
  - The runtime takes the decision of the iteration space distribution using the ***run-sched-var*** internal control variable value
  - This variable can be modified using the **OMP_SCHEDULE** environment variable

Reductions

```
int sum=0, A[MAX], int i;

for (i=0;i< MAX; i++)
{
 sum + = A[i];
}
```

- In the above example, ‘sum’ can be partially computed in parallel
- But if the ‘sum’ variable is shared among the threads, there is a risk of race condition
- If ‘sum’ is declared lastprivate, at the end of the ‘for/do’ region its value will be the value of ‘sum’ computed by the thread which will execute the last iteration; not the sum of each thread ‘sum’ variable

The ‘reduction’ Clause (1/3)

- A reduction is an associative operation on a shared variable
- Syntax of the **reduction** clause

  reduction(op:list)

  - Where ‘op’ is an operation and ‘list’ is a list of variables
- Principle
  - Each thread will have a private copy of the variable and will compute a partial result
  - The threads synchronize at the end of the construct to compute the final result

The ‘reduction’ Clause (2/3)

- C/C++ supported operations:
  - arithmetic: +, -, *
  - logical: &, |, ^, &&, ||
  - other: min, max

- FORTRAN supported operations :
  - arithmetic: +, -, *
  - logical: .and., .or., .eqv., .neqv.
  - intrinsic: max, min, iand, ior, ieor

The ‘reduction’ Clause (3/3)

- Example of reduction

```
int sum = 0;
#pragma omp for reduction(+:sum)
for (i=0;i< MAX; i++)
{
  sum + = A[i];
}
```

- The reduction clause can be used with parallel construct

```
int sum = 0;
#pragma omp  parallel reduction(+:sum) num_threads(3)
{
  sum + = 4;
}
// here ‘sum’ has the value: 12 = 4 * 3 (number of threads)
```

The ‘Workshare’ Construct (1/2)

- This construct exists only in Fortran
- As the ‘for/do’ construct, the **workshare** construct allows worksharing between threads of a parallel region
- It divides the execution of the contained structured code block into units of work
- Each unit of work is executed an unique time by one of the threads
- Syntax

```
!$omp workshare
  structured-block
!$omp end workshare [nowait]
```

The ‘Workshare’ Construct (2/2)

- The workshare region only supports
  - array affectations (several units of work)
  - scalar affectations (one unit of work)
  - ‘FORALL’ declarations and constructs (several units of work)
  - ‘WHERE’ declarations and constructs (several units of work)
  - ‘atomic’ constructs (one unit of work)
  - ‘critical’ constructs (one unit of work)
  - ‘parallel’ constructs (one unit of work)

Master and Single Constructs

17-10-2016

The ‘master’ Construct

- The **master** construct specifies a structured code block which only the master thread will execute

- Syntax

```
#pragma omp master
{
  //Only the master thread will execute this block
}
```

- There is no thread synchronization before nor after a **master** region

- In the same parallel region, race conditions between two master regions are impossible

The ‘single’ Construct

- The single construction specifies a structured code block which only one thread will execute
- There is an implied synchronization barrier between threads at the end of the single region
- Syntax

```
#pragma omp single [clause[[,] clause] ...]
{
  // only one thread executes this block

} //implied barrier
```

- Available clauses
  - private(*list*), firstprivate(*list*), copyprivate(*list*), nowait

The ‘copyprivate’ clause

- The **copyprivate** clause only exists for single constructs
- it propagates the value of a private variable to all the threads at the end of a single region
- The **copyprivate** clause cannot be used with the **nowait** clause

```
#pragma omp parallel
{
  int a = 12;
 #pragma omp single copyprivate(a)
 {
   a = 42;
 }
  //here ‘a’ has the value 42 for all the threads of the parallel region
}
```

The Internal Control Variables

17-10-2016

The Internal Control Variables

- An OpenMP implementation must behave as if internal control variables (ICVs) drive the behavior of a OpenMP program
- These variables stocks information such as: the thread number for a future parallel region, the loop scheduling of for/do region, …
- These variables are initialized with default value and can be modified by OpenMP environment variables or API
- In an OpenMP program, the value of some of these variables can be retrieve using the API

The Internal Control Variables

ICV	Associated environnent variable	Initial value
dyn-var	**OMP_DYNAMIC**	Implementation dependent
nest-var	**OMP_NESTED**	false
nthreads-var	**OMP_NUM_THREADS**	Implementation dependent
run-sched-var	**OMP_SCHEDULE**	Implementation dependent
def-sched-var	-	Implementation dependent
bind-var	**OMP_PROC_BIND**	Implementation dependent
stacksize-var	**OMP_STACKSIZE**	Implementation dependent
wait-policy-var	**OMP_WAIT_POLICY**	Implementation dependent
thread-limit-var	**OMP_THREAD_LIMIT**	Implementation dependent
max-active-levels-var	**OMP_MAX_ACTIVE_LEVELS**	Implementation dependent
active-levels-var	-	zero
levels-var	-	zero
place-partition-var	**OMP_PLACES**	Implementation dependent
cancel-var	**OMP_CANCELLATION**	false
default-device-var	**OMP_DEFAULT_DEVICE**	Implementation dependent

Best Practices and Performances

17-10-2016

Maximize Parallel Regions (1/2)

- Sequence of ‘omp for’ regions

```
...

#pragma omp parallel for
{
 for(){...} //loop1
}

#pragma omp parallel for
{
 for(){...} //loop2
}

#pragma omp parallel for
{
 for(){...} //loop3
}
...
```

```
...

#pragma omp parallel
{

 #pragma omp for
 for(){...} //loop1

#pragma omp for
 for(){...} //loop2

#pragma omp for
 for(){...} //loop3

}
...
```

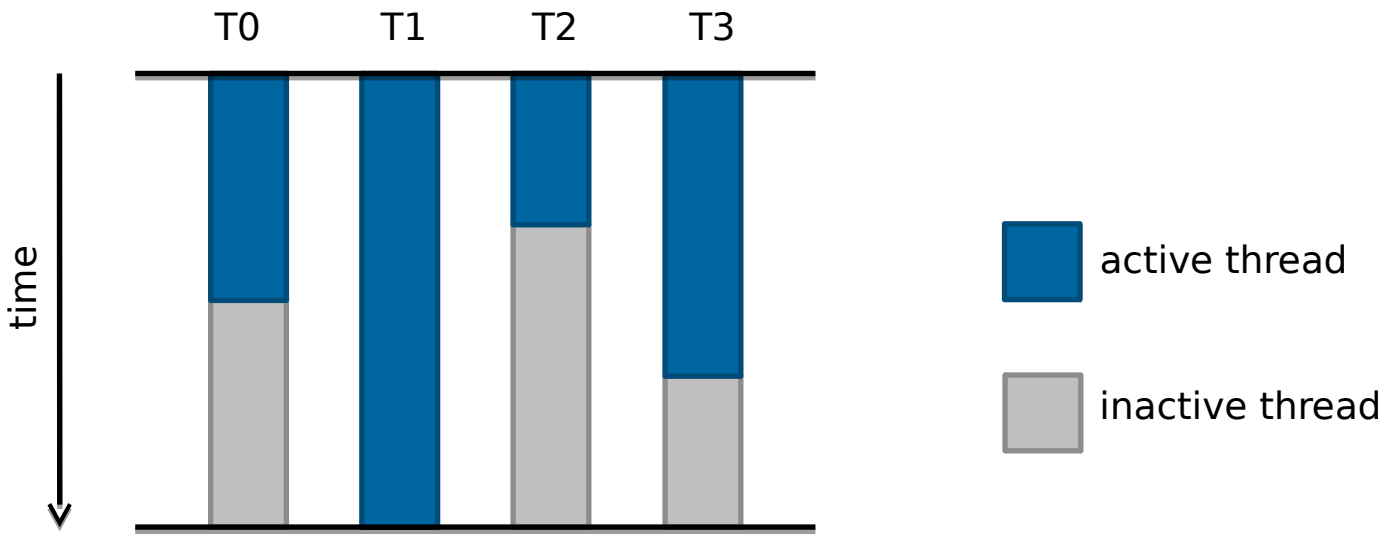
- [illegible]

Maximize Parallel Regions (2/2)

- Minimize the cost of creation and destruction of threads
  - Runtimes tends to dynamically optimize it
- Decrease the risk to have the operating system migrate threads to other cores
  - Can be fully avoided with the support of thread affinity
- Prerequisite in worksharing region for the use of the nowait clause
  - Each *parallel* region ends with an implied barrier which cannot be removed

Load Balancing (1/3)

- Load balancing is a critical aspect of performance when using OpenMP
  - not really important on regular problem such as addition of vectors
  - for irregular problem, a bad distribution of work implies a loss of performance



Load Balancing (2/3)

- How to tweak the load balancing
  - useless implied barrier can be removed
    - *nowait* clause: an unoccupied thread may start an other task without waiting for the other threads
  - several possible scheduling for ‘for/do’ loops
    - static, dynamic, guided
    - size of chunks
  - *untied* clause for explicit tasks
    - an unoccupied thread can resume a pending task started by an other thread.
  - …

Directives & Optimizations

- **Adding directives may prevent the compiler to perform some optimizations**

- Without OpenMP, the compiler can fused the two loops and factorize the read of *B[i]*

- The *for* region may prevent this optimization
  - Add an implied barrier which modify the semantic
  - With a *nowait* clause, the scheduling is still an issue for the optimization (ex: dynamic)

```
...
#pragma omp parallel shared(A,B,C)
{

 #pragma omp for
  for(i=0; i<N; i++)
    A[i] = B[i] *2;

 #pragma omp for
  for(i=0; i<N; i++)
     C[i] = B[i] +3;

}
...
```

Thanks

For more information please contact:
Cyril Mazauric
Cyril.Mazauric@atos.net

29-09-2015