# Training @ CINES:

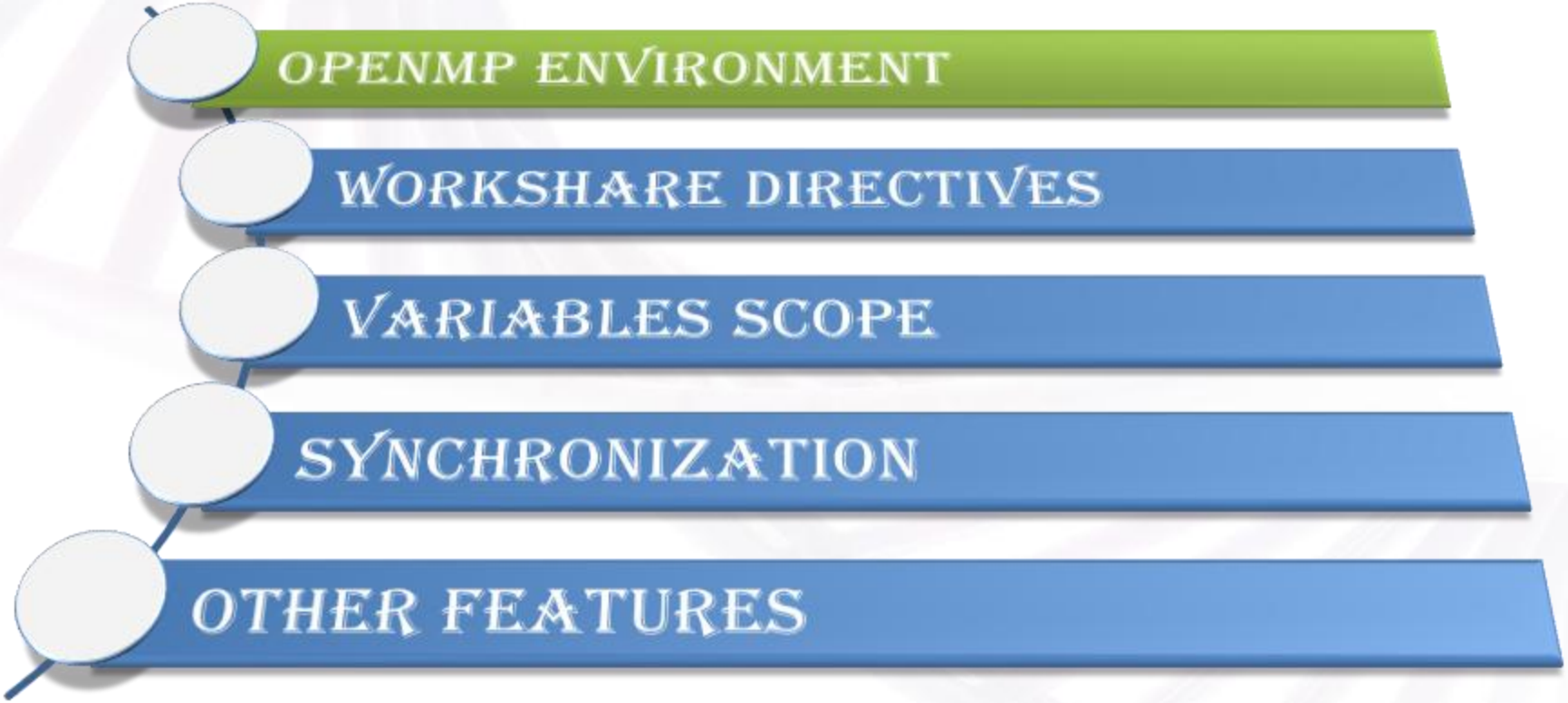
# OpenMP

Johanne Charpentier & Gabriel Hautreux
charpentier@cines.fr hautreux@cines.fr

# Summary

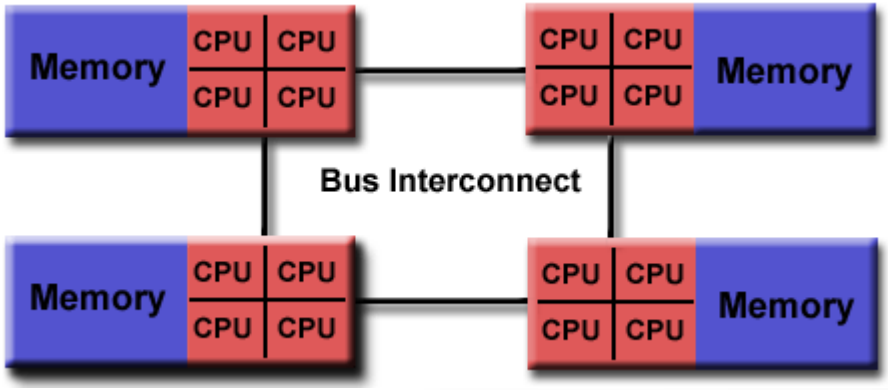

OPENMP ENVIRONMENT
WORKSHARE DIRECTIVES
VARIABLES SCOPE
SYNCHRONIZATION
OTHER FEATURES

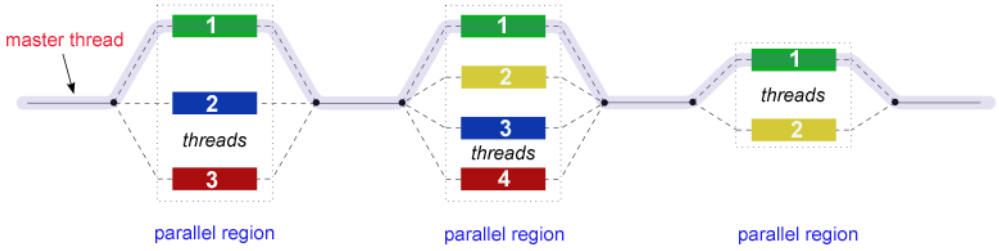
# Summary


OPENMP ENVIRONMENT
WORKSHARE DIRECTIVES
VARIABLES SCOPE
SYNCHRONIZATION
OTHER FEATURES

# What is OpenMP?

**OpenMP is an API for shared memory application**




## Features

- Support C/C++ and Fortran
- Uses the thread fork/join model
- Scalable on UMA and NUMA architectures (Shared memory model)
- Compiler directives
- Environment variables

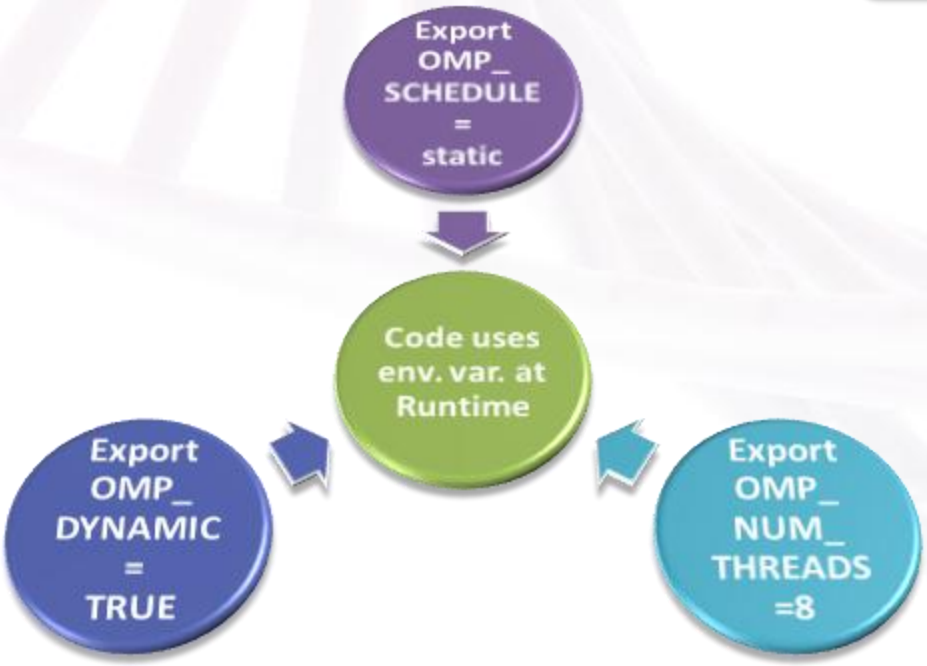
# Compilation

**Compiling with OpenMP is straightforward**

- Include the OpenMP header (for C/C++)
  #include<omp.h>
- Add some OpenMP directives
- Uses the right compiler option:
  Intel: -openmp
  Gnu : -fopenmp
- Set your environment variable:
  OMP_NUM_THREADS
  to define the number of threads you want to use (default is the number of core available) Your code is now using OpenMP

# Environment

**OpenMP can be used with environment variables**




Those value can also be set inside the code

- OpenMP provides functions such as:
- omp_set_num_threads()
- omp_get_num_threads()
- omp_get_max_threads()
- …

Have a look at the OpenMP refcard for more funtionalities!

(http://openmp.org/mp-documents/OpenMP-4.0-C.pdf)

# Directives

**OpenMP implementation is based on compiler directives**

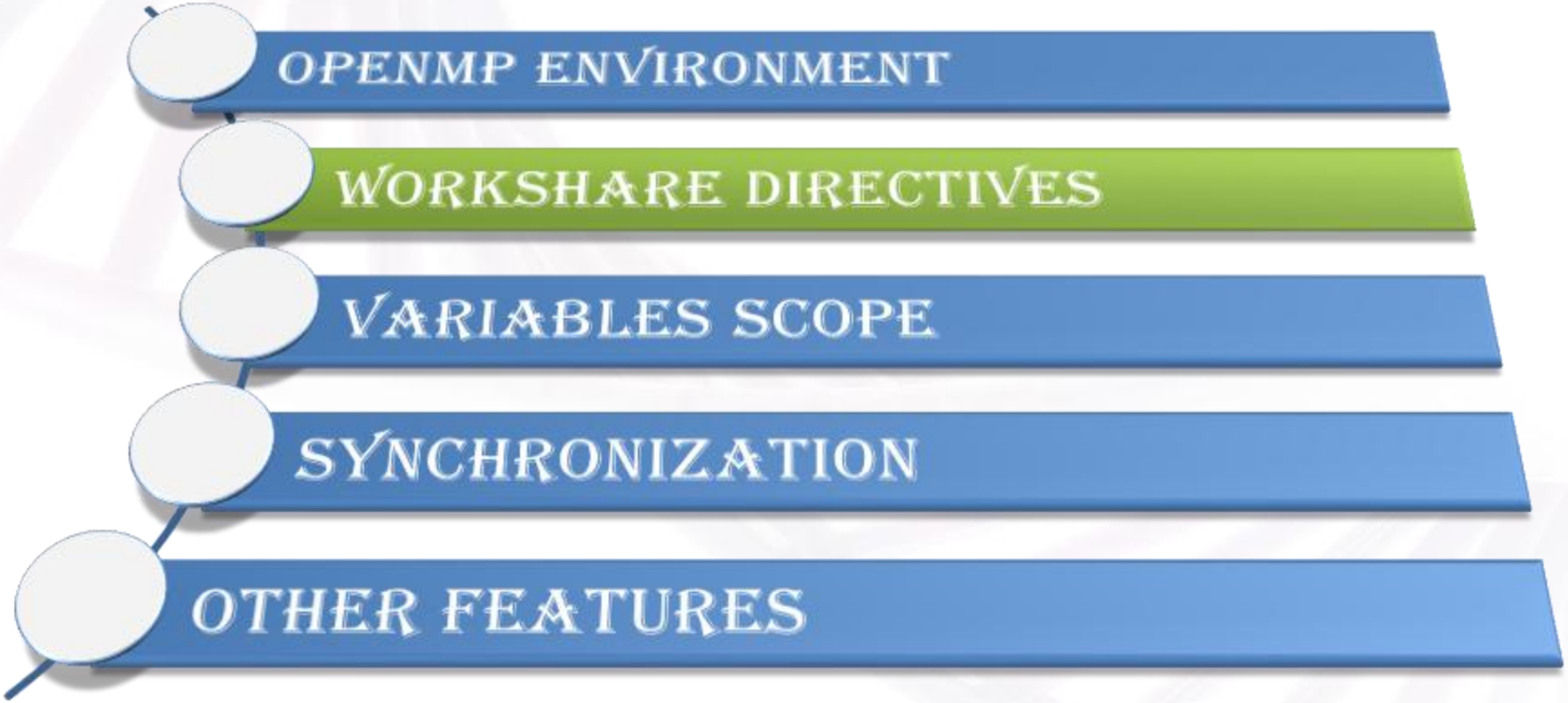
**Fortran:** directive syntax

```
$!OMP DIRECTIVE [CLAUSES(variables, schedule,…),…]
« execute parallel region »
$!OMP END DIRECTIVE
```

**C/C++:** directive (pragma) syntax

```
#pragma omp directive [clauses(variables, schedule,…),…]
{
« execute parallel region »
}
```

# Summary


OPENMP ENVIRONMENT
WORKSHARE DIRECTIVES
VARIABLES SCOPE
SYNCHRONIZATION
OTHER FEATURES

# Parallelism

**How would you do to parallelise your code?**

# Parallelism

**How would you do to parallelise your code?**

- Divide arrays computation between all the threads ?
- Make different threads compute different things ?
- Define a pool of things to do and perform it ASAP ?

# Parallelism

**How would you do to parallelise your code?**

- Divide arrays computation between all the threads ?
- Make different threads compute different things ?
- Define a pool of things to do and perform it ASAP ?

TRUE

# Parallel Region

## A parallel region launches threads

```
#pragma omp parallel
{
  printf(« HelloWorld! »);
}
```

This code will have the following output:

```
HelloWorld!
HelloWorld!
HelloWorld!
HelloWorld!
HelloWorld!
HelloWorld!
…
```

**You can use as many OpenMP directives as you want inside a parallel region!**

# OpenMP for loops

**Parallelise compute intensive loops**

## Serial version

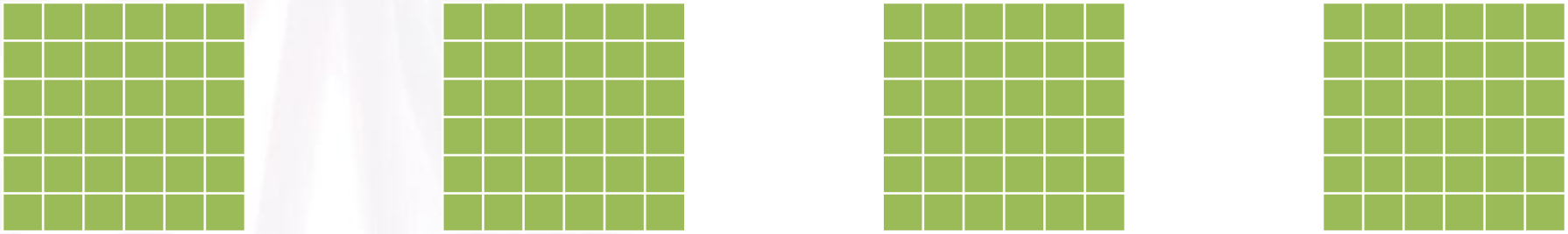
```
for(i=0;i<n;i++)
{
  for(j=0;j<n;j++)
  {
    a[i][j]=b[i][j]
       +c[i][j]*d[j][i];
  }
}
```

## Parallel version

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

# Matrix FMA

Serial



A = B + C x D

Parallel

# OpenMP for loops

**Why should we never parallelize the inner loop?**

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

# OpenMP for loops

**Why should we never parallelize the inner loop?**

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

Huge overhead due to thread creation/destruction!

# OpenMP sections

**Parallelise non dependant parts of your code**

## Serial version

```
for(i=0;i<n;i++)
{
  a[i]=b[i]*f(i);
}

for(i=0;i<n;i++)
{
  c[i]=g(b[i]);
}
```

## Parallel version

```
#pragma omp parallel sections
{
#pragma omp section
  for(i=0;i<n;i++)
  {
    a[i]=b[i]*f(i);
  }
#pragma omp section
  for(i=0;i<n;i++)
  {
    c[i]=g(b[i]);
  }
}
```

# OpenMP single

**Enable some parts to be computed by only one thread**

## Serial version

```
for(i=0;i<n;i++)
{
  a[i]=b[i]*c(i);
}
printf(« Switching a & c »);
 a <-> c ;
for(i=0;i<n;i++)
{
  c[i]=b[i]*a[i];
}
```

## Parallel version

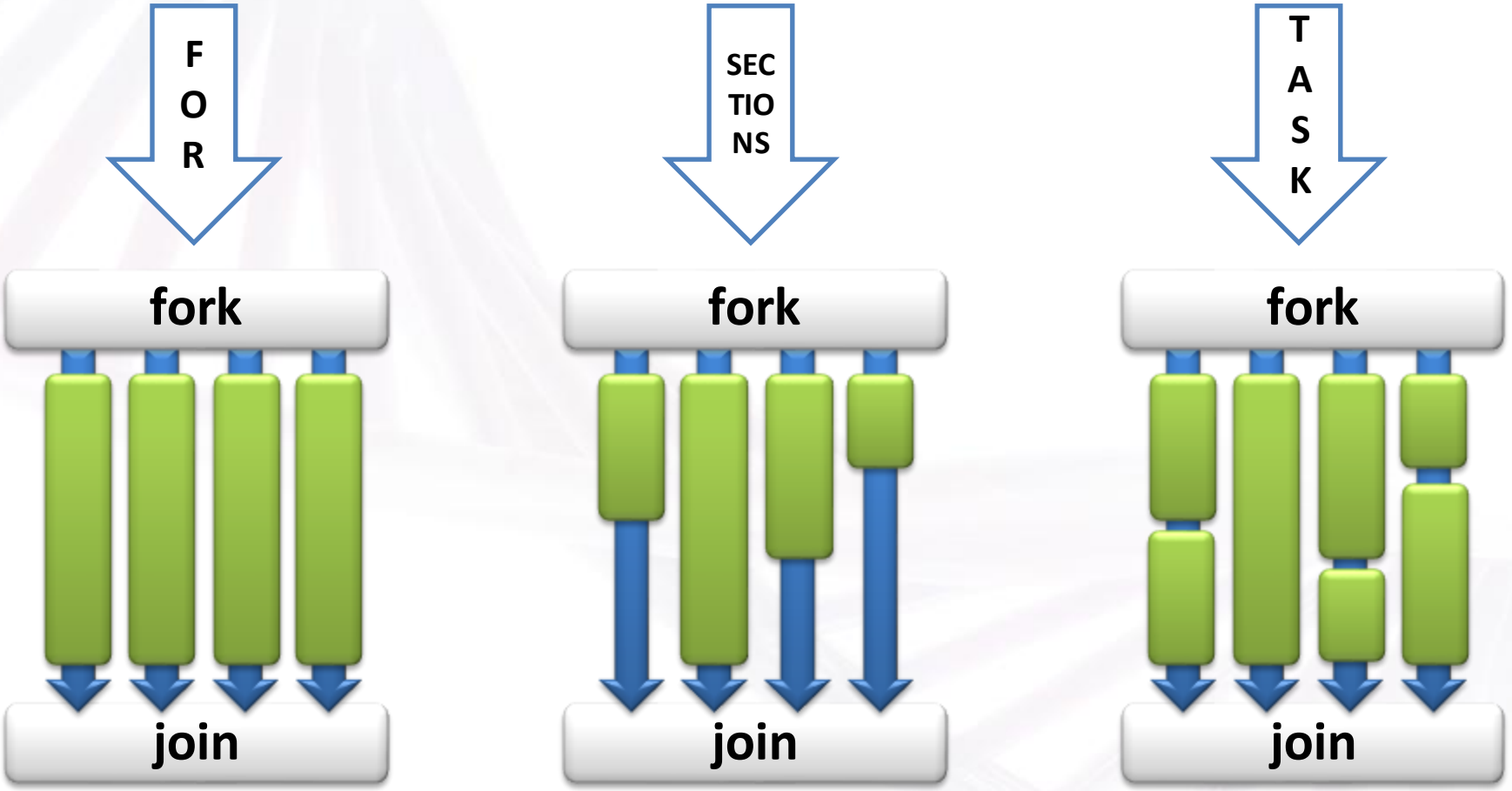
```
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++)
  a[i]=b[i]*c(i);
#pragma omp single
{
printf(« Switching a & c »);
 a <-> c ;
}
#pragma omp for
for(i=0;i<n;i++)
  c[i]=b[i]*a[i];
}
```

# Bonus: tasks

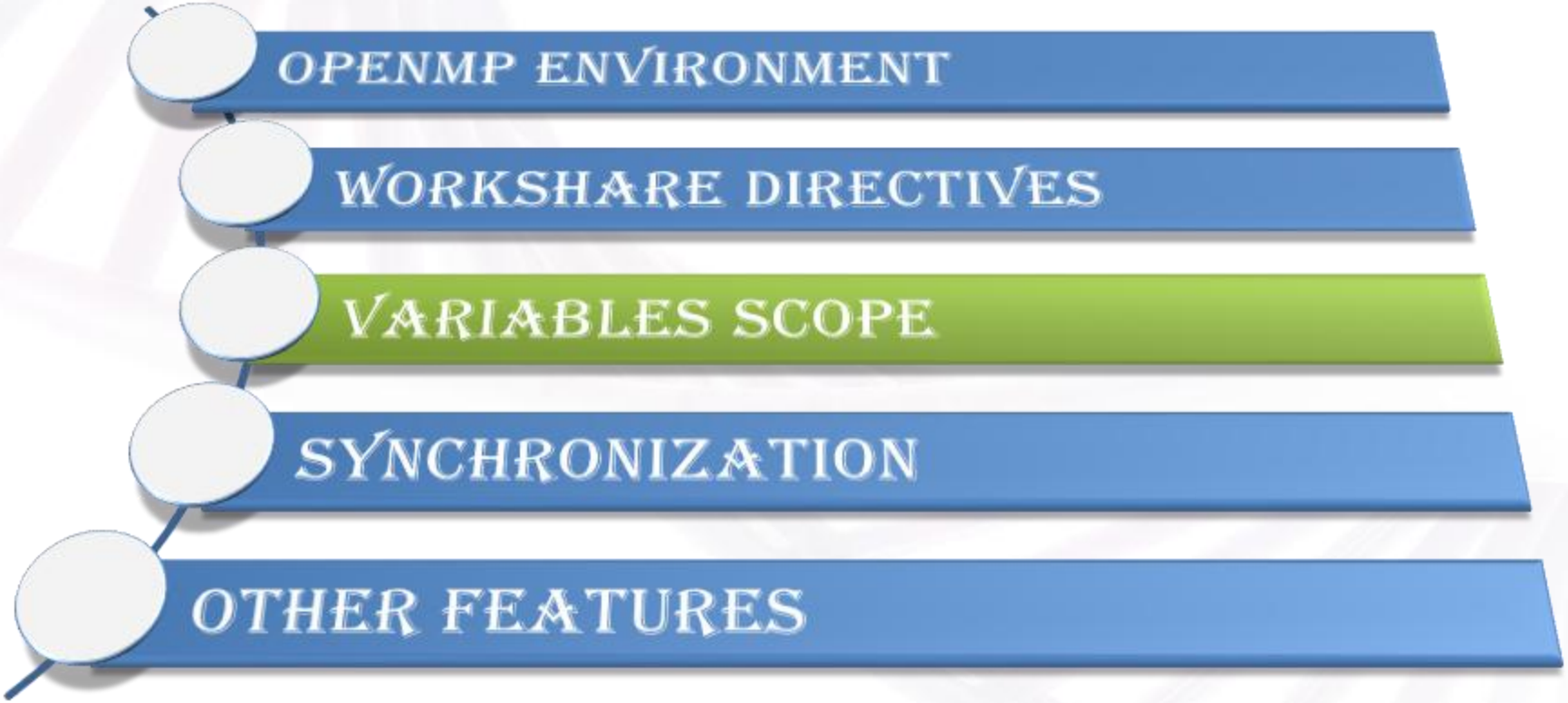
**Close to sections, but creates a pool of non dependant tasks**

```
#pragma omp parallel
{
  #pragma omp single
  {
    for(task = 0; task < nTask; ++task)
    {
      range = task;
      #pragma omp task firstprivate(range)
      {
        myFunction(range);
      }
    }
  }
}
```

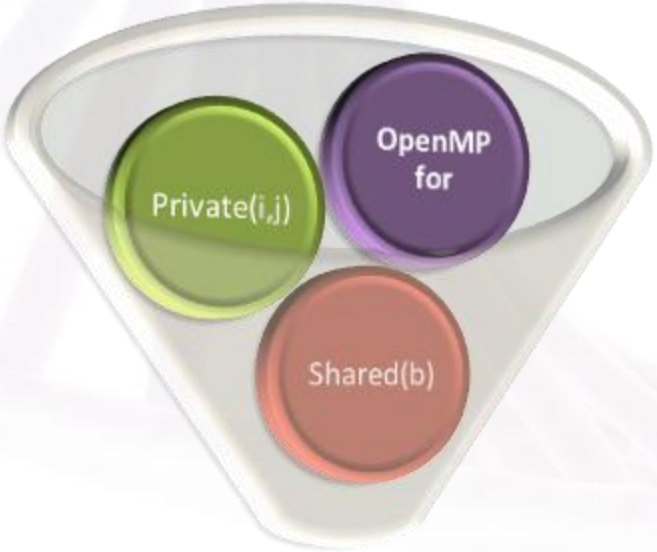
# Workshare



# Summary


OPENMP ENVIRONMENT
WORKSHARE DIRECTIVES
VARIABLES SCOPE
SYNCHRONIZATION
OTHER FEATURES

# Variables Scope




**Variables within parallel region can be used in several different ways**

**Many options for variable scopes**

- PRIVATE
- SHARED
- DEFAULT
- FIRSTPRIVATE
- LASTPRIVATE
- COPYIN
- COPYPRIVATE
- REDUCTION

# Private clause

**Variables in Private are private to each thread**

```
#pragma omp parallel private(rank)
{
  rank = omp_get_thread_num ();
  printf(« I am thread number: %d », rank);
}
```

This code will have the following output (for 4 threads):

```
I am thread number: 1
I am thread number: 2
I am thread number: 3
I am thread number: 4
```

If rank was not private, you could have had (for instance):

```
I am thread number: 2
I am thread number: 2
I am thread number: 2
I am thread number: 1
```

# Public clause

**Variables in shared are shared by all threads**

```
Int b = 10;
#pragma omp parallel private(rank) shared(b)
{
  rank = omp_get_thread_num ();
  printf(« I am thread number: %d, b= %d », rank);
}
```

This code will have the following output (for 4 threads):

```
I am thread number: 1, b=10
I am thread number: 2, b=10
I am thread number: 3, b=10
I am thread number: 4, b=10
```

If b was not defined as shared, it still would have been shared.
In fact, default is shared.
You can change the default by adding the clause:
- default(private)

- default(none)

# Firstprivate clause

**Firstprivate are private variables with their initial values**

```
Int b = 10;
#pragma omp parallel private(rank) firstprivate(b)
{
  rank = omp_get_thread_num ();
  printf(« I am thread number: %d, b= %d », rank);
}
```

This code will have the following output (for 4 threads):

```
I am thread number: 1, b=10
I am thread number: 2, b=10
I am thread number: 3, b=10
I am thread number: 4, b=10
```

If b was defined as private, the output would be:

```
I am thread number: 1, b=0
I am thread number: 2, b=0
I am thread number: 3, b=0
I am thread number: 4, b=0
```

# Reduction

**Reduction will perform an operation at the end of a region**

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

printf(« the product of the elts of a is: %d, product);
```

***x = x op expr***
***x = expr op x***
***x binop = expr***

***x*** is a scalar variable in the list
***expr*** is a scalar expression that does not reference ***x***
***op*** is not overloaded, and is one of +, *, -, /, &, ^, |, &&, ||
***binop*** is not overloaded, and is one of +, *, -, /, &, ^, |

# Summary


OPENMP ENVIRONMENT
WORKSHARE DIRECTIVES
VARIABLES SCOPE
SYNCHRONIZATION
OTHER FEATURES

# Master

**The region is only executed by the master thread**

```
#pragma omp parallel
{
  #pragma omp master
    printf(« I am the master thread »);

  printf(« I am a slave »);
}
```

Only the master thread will write:

```
I am the master thread
```

# Barrier

**Wait for all the threads to reach the barrier to continue**

```
#pragma omp parallel
{
  if(omp_get_num_thread())=1
    b=10;
  if(omp_get_num_threads())=0
    a=5;
  #pragma omp barrier

  printf(« a=%d,b=%d »,a,b);
}
```

With the barrier, the output is:

a=5, b=10

Without the barrier, the output is random

a= ?, b= ?

# Critical

**Critical section is performed by one thread at a time**

```
int b=0,c=1;
#pragma omp parallel shared(b,c) num_threads(3)
{ int a=10;
#pragma omp critical
 {
   b=b+a;
   c=c*a;
 }
}
  printf(« a=%d, b=%d);
```

With the critical, the output is:

b=30, c=1000

Critical does not include any implicit barrier

# Atomic

**Atomic section is performed by one thread at a time**

```
int b=0,c=1;
#pragma omp parallel shared(b,c) num_threads(3)
{ int a=10;
  #pragma omp atomic
     b=b+a;
  #pragma omp atomic
     c=c*a;
}
  printf(« a=%d, b=%d);
```

***x = x op expr***
***x = expr op x***
***x binop = expr***

***x*** is a scalar variable in the list
***expr*** is a scalar expression that does not reference ***x***
***op*** is not overloaded, and is one of +, *, -, /, &, ^, |, &&, ||
***binop*** is not overloaded, and is one of +, *, -, /, &, ^, |

# Summary


OPENMP ENVIRONMENT
WORKSHARE DIRECTIVES
VARIABLES SCOPE
SYNCHRONIZATION
OTHER FEATURES

# Other features

**A lot of other features are available in OpenMP**



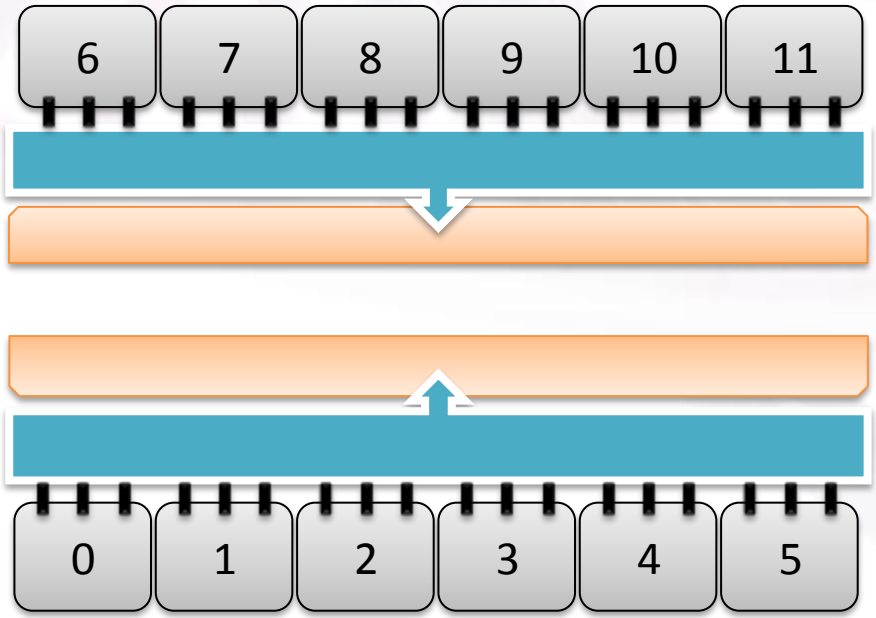
## Here are some OpenMP features

- Nested parallelism
- Offloading ? (in future releases of the compilers)
- Threads binding
- Stack size per thread definition
- …

# Thread placement

**Use KMP_AFFINITY to define your binding options**

KMP_AFFINITY=compact,0,0

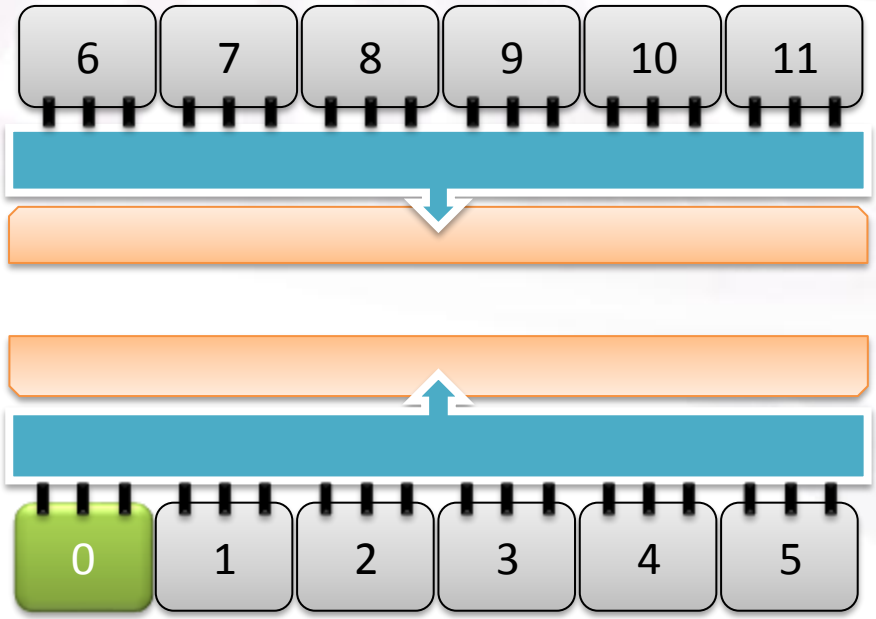



- Threads are as close as possible (compact)
- There is no offset (0)
- The first thread is binded to CPU number 0
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**1st thread is binded to cpu 0**

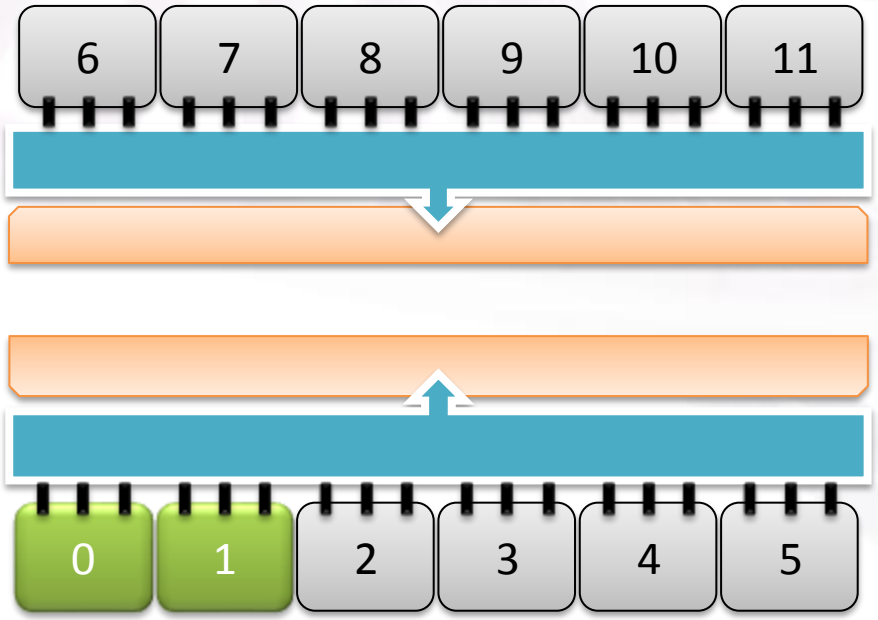
KMP_AFFINITY=compact,0,0




- Threads are as close as possible (compact)
- There is no offset (0)
- The first thread is binded to CPU number 0
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**2nd thread is binded to cpu 1**

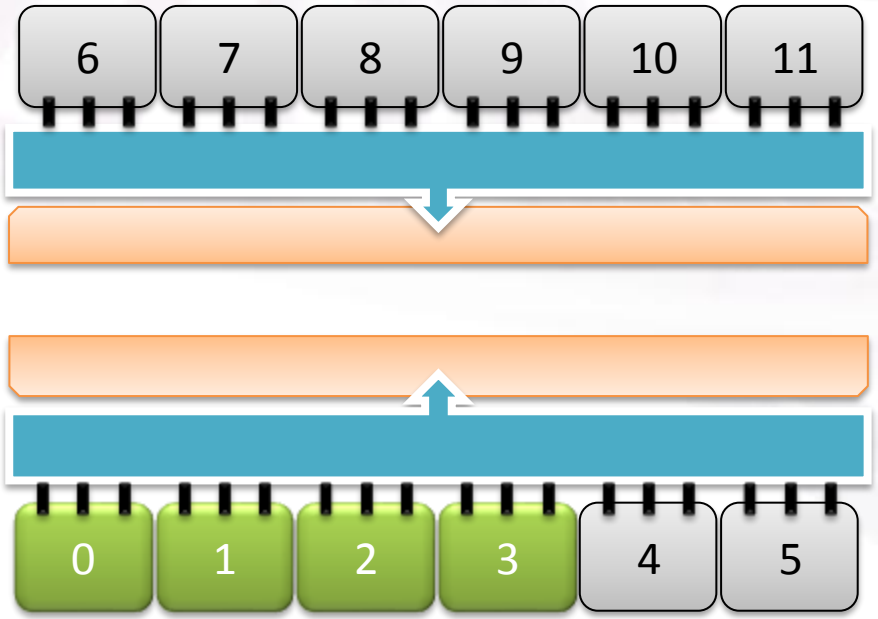



KMP_AFFINITY=compact,0,0

- Threads are as close as possible (compact)
- There is no offset (0)
- The first thread is binded to CPU number 0
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**3rd and 4th thread are binded to cpus 2 and 3**

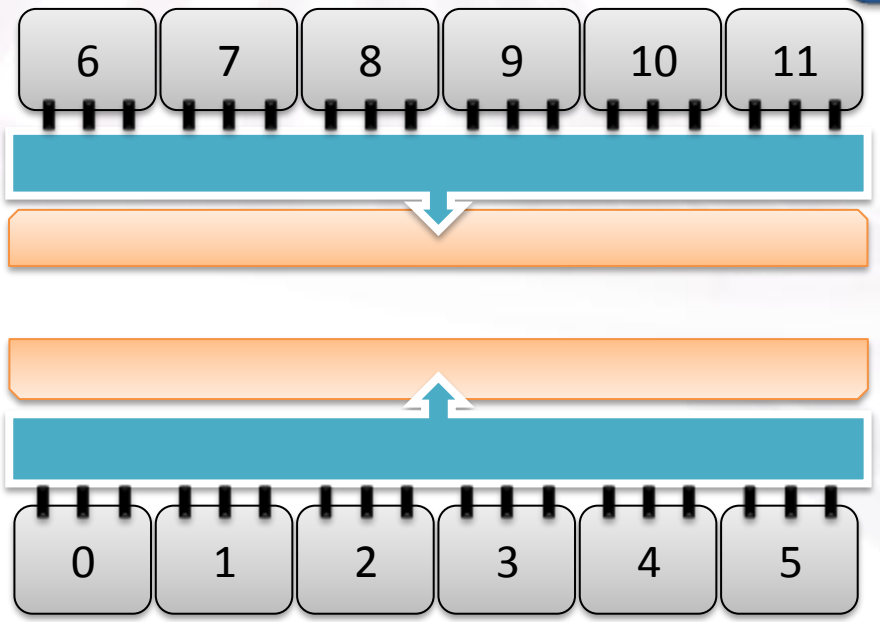



KMP_AFFINITY=compact,0,0

- Threads are as close as possible (compact)
- There is no offset (0)
- The first thread is binded to CPU number 0
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**Use KMP_AFFINITY to define your binding options**

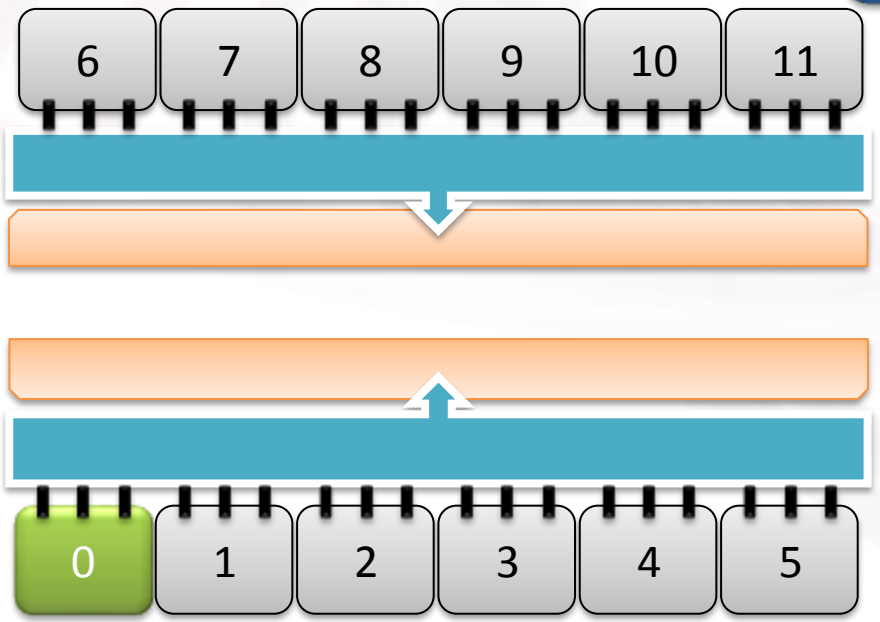



KMP_AFFINITY=scatter,0,0

- Threads are scattered among sockets
- There is no offset (0)
- The first thread is binded to CPU number 0
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**1st thread is binded to cpu 0**

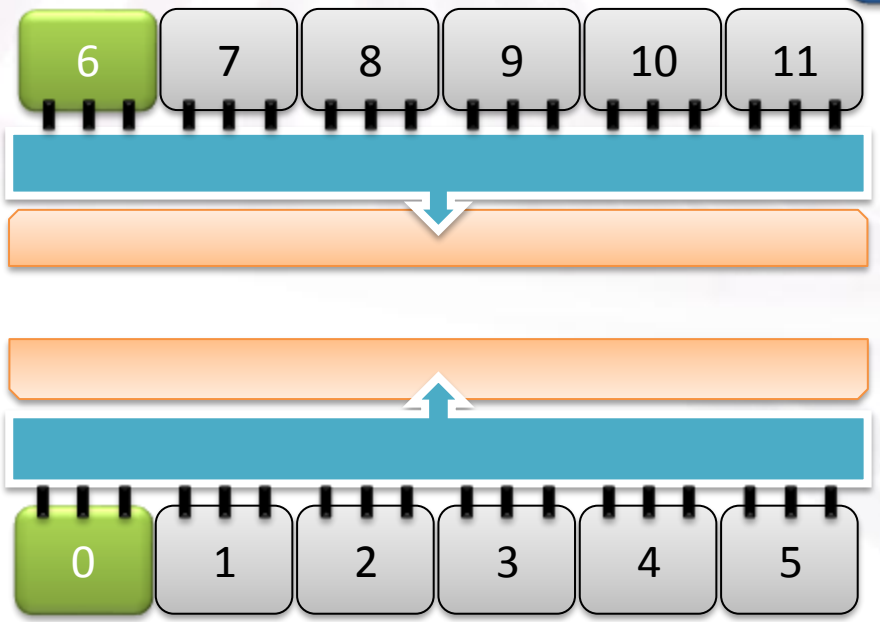



KMP_AFFINITY=scatter,0,0

- Threads are scattered among sockets
- There is no offset (0)
- The first thread is binded to CPU number 0
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**2nd thread is binded to cpu 6**

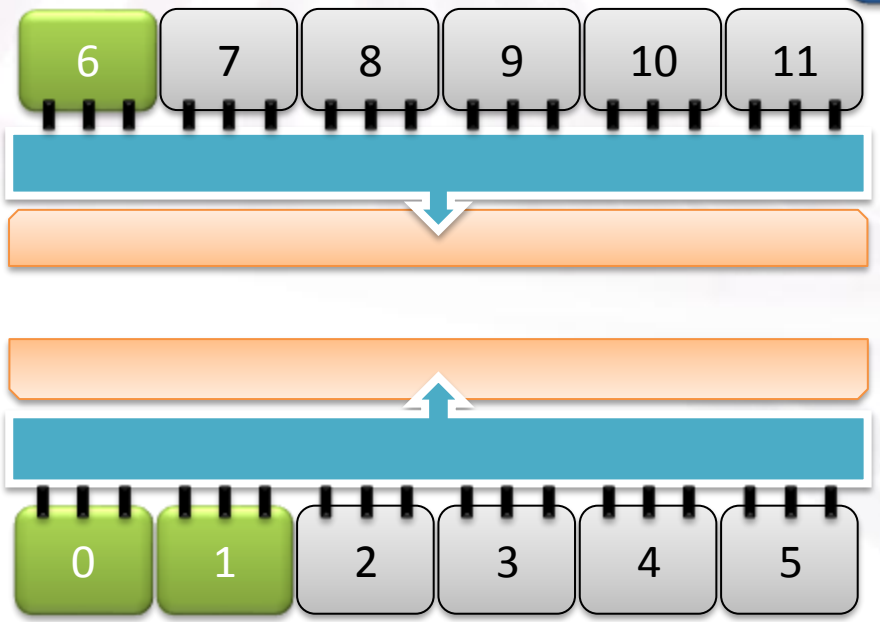



KMP_AFFINITY=scatter,0,0

- Threads are scattered among sockets
- There is no offset (0)
- The first thread is binded to CPU number 0
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**3rd thread is binded to cpu 1**

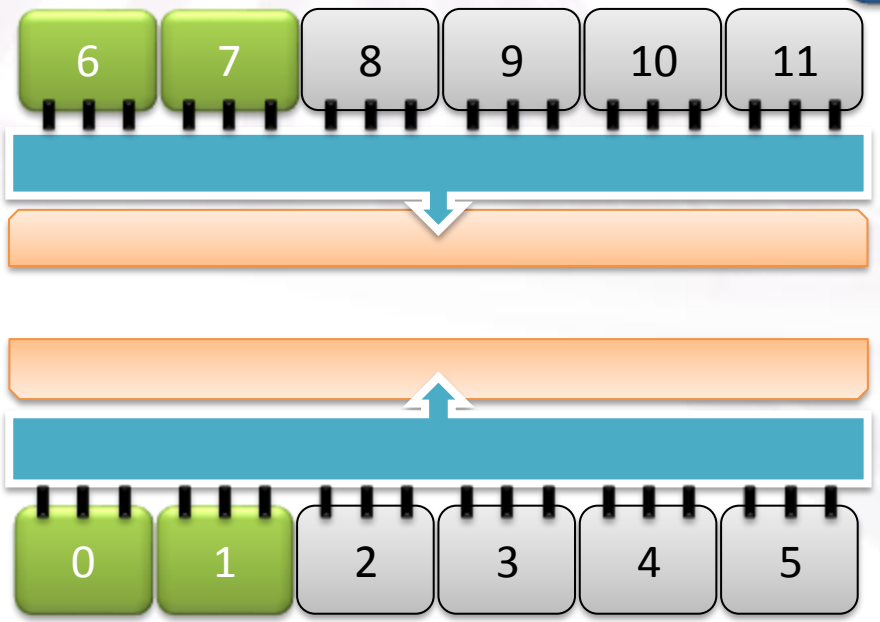



KMP_AFFINITY=scatter,0,0

- Threads are scattered among sockets
- There is no offset (0)
- The first thread is binded to CPU number 0
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**4th thread is binded to cpu 7**



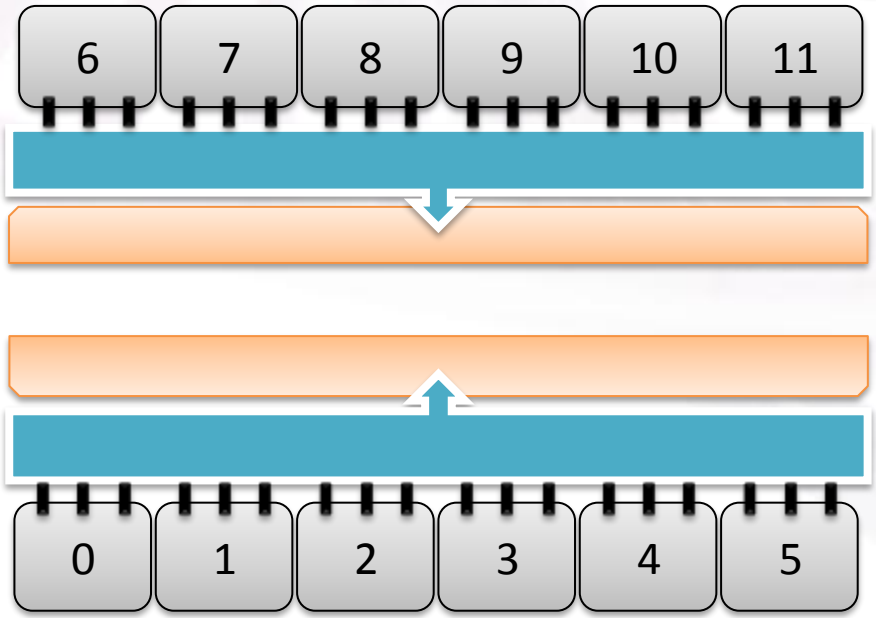

KMP_AFFINITY=scatter,0,0

- Threads are scattered among sockets
- There is no offset (0)
- The first thread is binded to CPU number 0
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**Use KMP_AFFINITY to define your binding options**

KMP_AFFINITY=compact,1,2

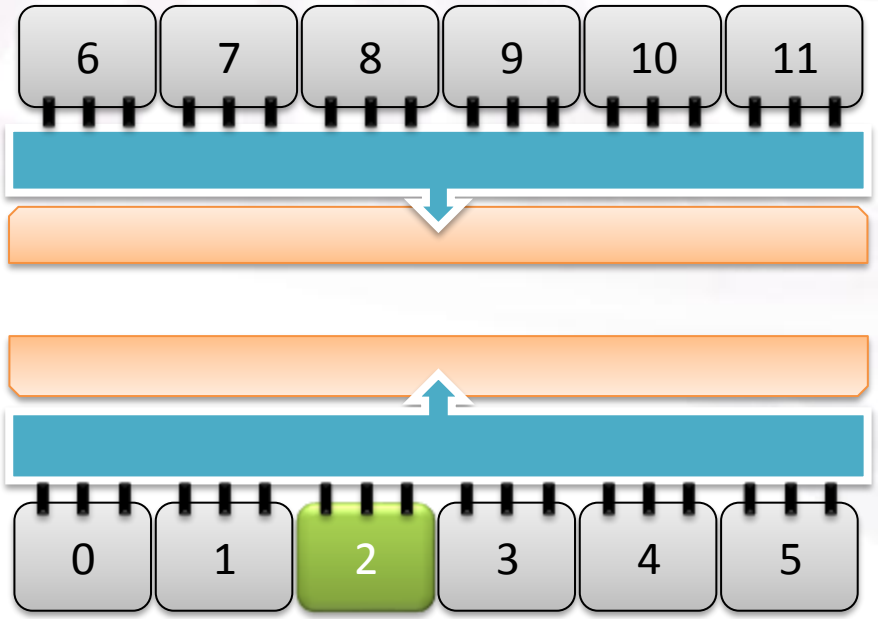



- Threads are as close as possible (compact)
- There is an offest of 1
- The first thread is binded to CPU number 2
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**1st thread is binded to cpu 2**

KMP_AFFINITY=compact,1,2

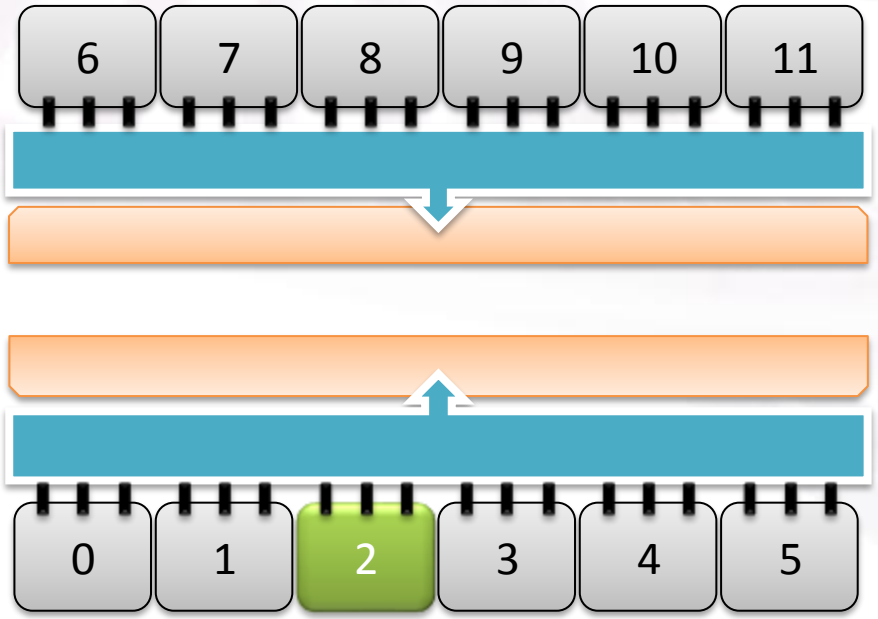



- Threads are as close as possible (compact)
- There is an offest of 1
- The first thread is binded to CPU number 2
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**1st thread is binded to cpu 2**

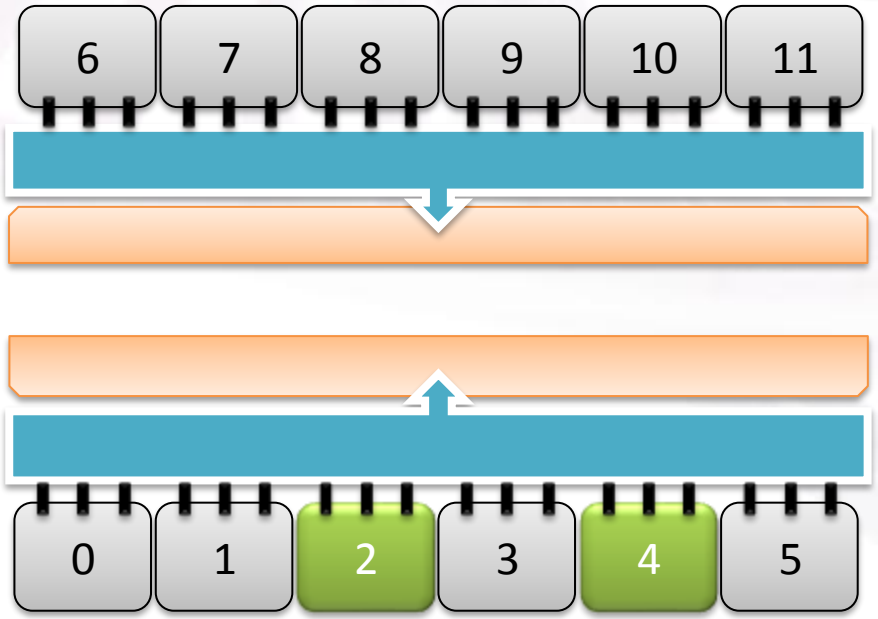
KMP_AFFINITY=compact,1,2




- Threads are as close as possible (compact)
- There is an offest of 1
- The first thread is binded to CPU number 2
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**2nd thread is binded to cpu 4**

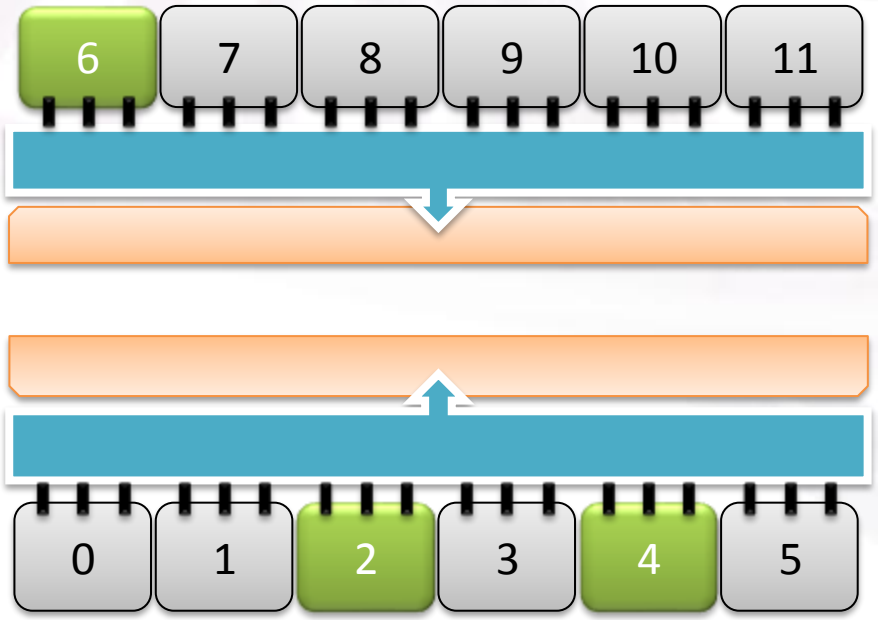



KMP_AFFINITY=compact,1,2

- Threads are as close as possible (compact)
- There is an offest of 1
- The first thread is binded to CPU number 2
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**3rd thread is binded to cpu 6**

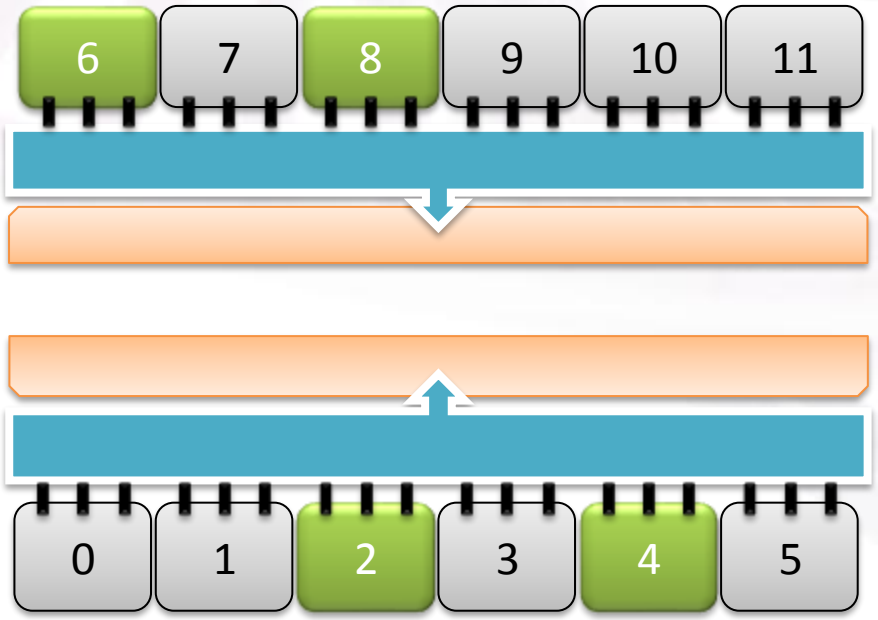



KMP_AFFINITY=compact,1,2

- Threads are as close as possible (compact)
- There is an offest of 1
- The first thread is binded to CPU number 2
- For OMP_NUM_THREADS=4 we have the following binding

# Thread placement

**4th thread is binded to cpu 8**




KMP_AFFINITY=compact,1,2

- Threads are as close as possible (compact)
- There is an offest of 1
- The first thread is binded to CPU number 2
- For OMP_NUM_THREADS=4 we have the following binding

# TP Time

**Have a look at the exercise sheet!**

# References

- https://computing.llnl.gov/tutorials/openMP/

In my opinion one of the best OpenMP tutorial

- OpenMP quick reference guide