# Introduction to OpenMP

March 29, 2017

## Setup Examples

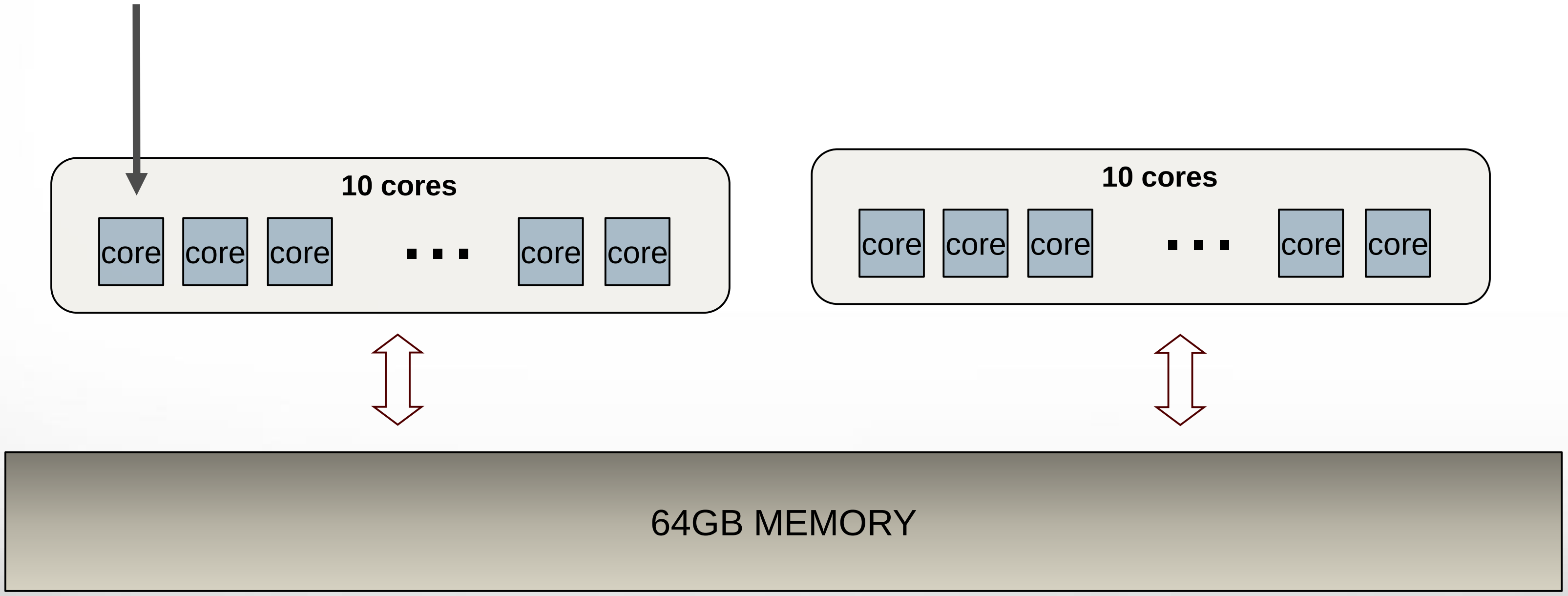
To copy the demos and examples to your local scratch, type:

**/general/public/training/openmp/setup.sh** **(ADA & CURIE)**
**/sw/local/training/openmp/setup.sh** **(TERRA)**

## Basic Computer Architecture

Each (terra/ada/curie) **NODE** contains multiple cores (28 on terra, 20 on ada, 16 on curie) and at at least 64GB (256 on curie) of **shared** memory (i.e. all cores have access to this memory)

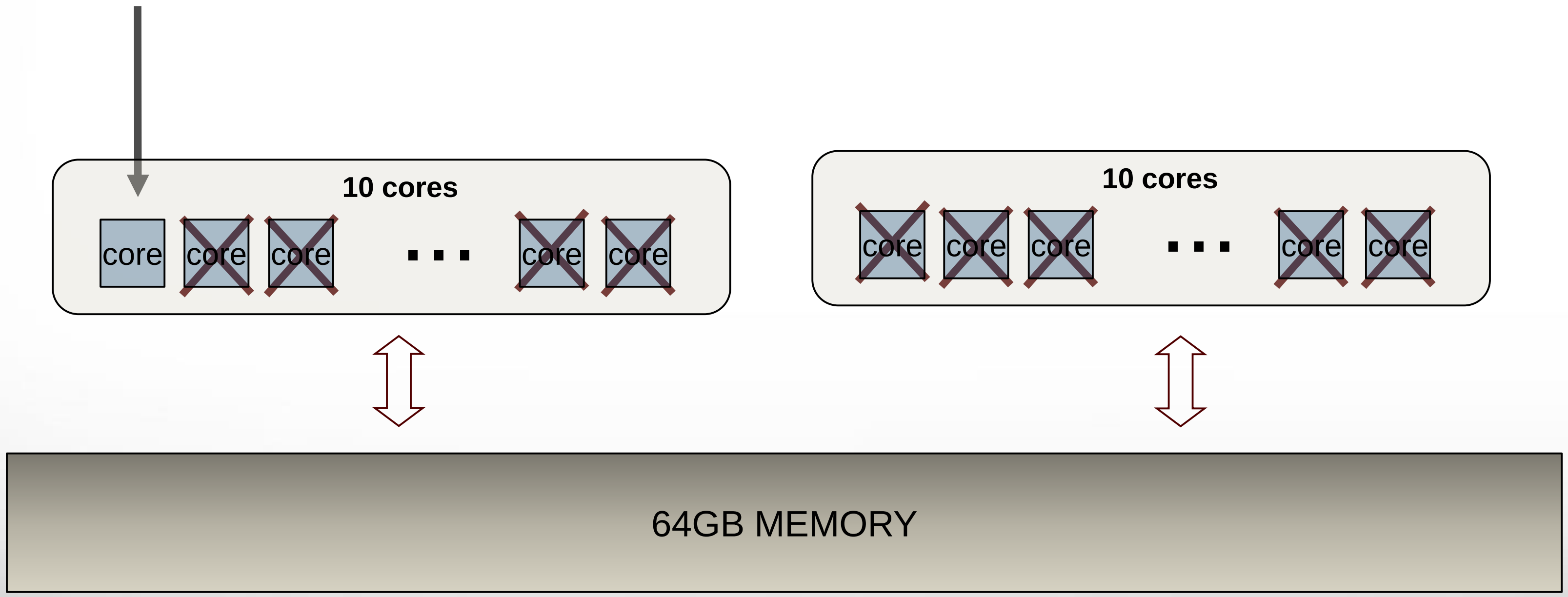
**NOTE: we use ada architecture for illustration below**



# Basic Computer Architecture

Each (terra/ada/curie) **NODE** contains multiple cores (28 on terra, 20 on ada, 16 on curie) and at at least 64GB (256 on curie) of **shared** memory (i.e. all cores have access to this memory)

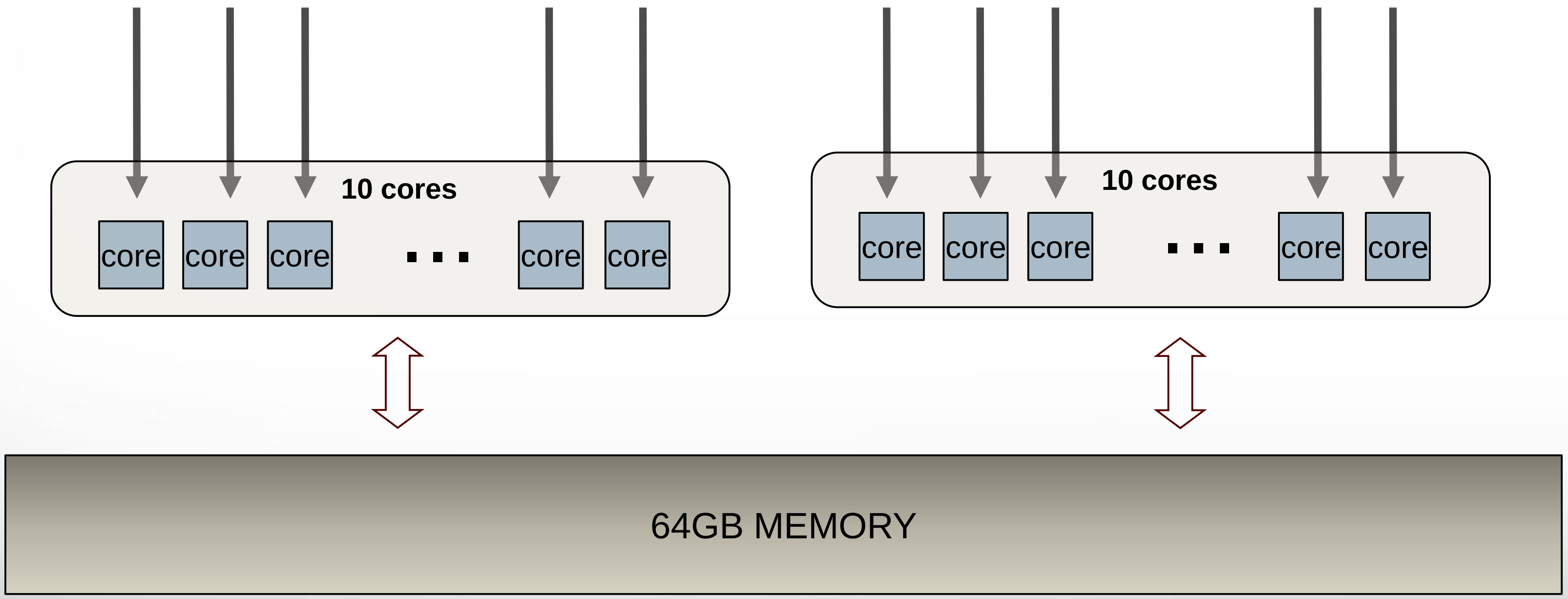
**NOTE: we use ada architecture for illustration below**



# Basic Computer Architecture

Each (terra/ada/curie) **NODE** contains multiple cores (28 on terra, 20 on ada, 16 on curie) and at at least 64GB (256 on curie) of **shared** memory (i.e. all cores have access to this memory)

**NOTE: we use ada architecture for illustration below**



# What is OpenMP?

Defacto standard API for writing ***shared memory*** parallel applications in C, C++, and Fortran

OpenMP API consists of:

- Compiler pragmas/directives
- Runtime subroutines/functions
- Environment variables

# OpenMP Directives/Pragmas

fortran directive format:

```
!$OMP DIRECTIVE   [clauses]
    :
!$OMP END DIRECTIVE
```

**Not case sensitive**

C/C++ pragma format is:

```
#pragma omp directive [clauses]
{
    :
}
```

**New line required**

**All OpenMP directives follow this format.**

# OpenMP HelloWorld

**SOURCE CODE:**

directive

```
program HELLO
!$OMP PARALLEL
print *,"Hello World"
!$OMP END PARALLEL
end program HELLO
```

pragma

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

int main() {
#pragma omp parallel
  {
    std::cout <<  "Hello World\n";
  }
  return 0;
}
```

**COMPILING:**

```
intel: ifort -openmp -o hi.x hello.f90
pgi:   pgfortran -mp -o hi.x hello.f90
gnu:   gfortran -fopenmp -o hi.x hello.f90
IBM:   xlf_r  -qsmp=omp -o hi.x hello.f90
```

```
intel: icpc -openmp -o hi.x hello.cpp
pgi:   pgcpp -mp  -o hi.x hello.cpp
gnu:   g++ -fopenmp -o hi.x hello.cpp
IBM:   xlc++_r  -qsmp=omp -o hi.x hello.cpp
```

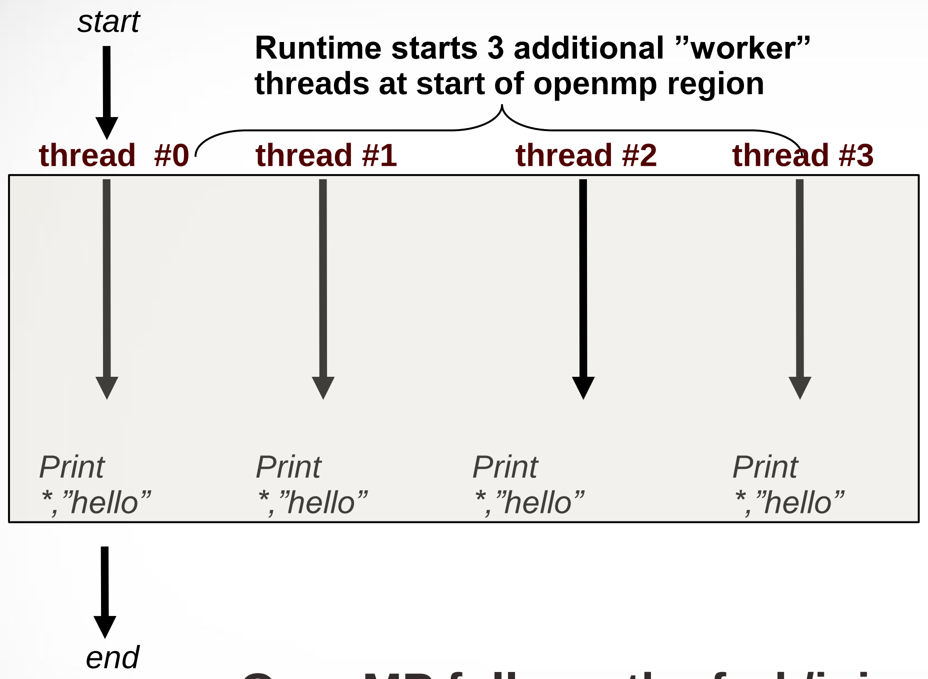
**RUNNING:**

```
export OMP_NUM_THREADS=4
./hi.x
```

environmental variable

NOTE: example hello world

# Fork/Join



```
program HELLO
!$OMP PARALLEL
print *,"Hello World"
!$OMP END PARALLEL
end program HELLO
```

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

**OpenMP follows the fork/join model:**

- OpenMP programs start with a single thread; the master thread (Thread #0)
- At start of parallel region master starts team of parallel "worker" threads (FORK)
- Statements in parallel block are executed in parallel by every thread
- At end of parallel region, all threads synchronize, and join master thread (JOIN)

**Implicit barrier. Will discuss synchronization later**

## Cores & Threads

**What are threads, cores, and how do they relate?**

- ➢ Thread is independent sequence of execution of program code
  - ✓ Block of code with one entry and one exit
- ➢ For our purposes a more abstract concept
- ➢ Unrelated to physical cores/CPUs
- ➢ OpenMP threads are mapped onto physical cores
- ➢ Possible to map more than 1 thread onto a core
- ➢ In practice best to have one-to-one mapping.

# Setting the number of Threads

case sensitive

- Environmental variable: **OMP_NUM_THREADS**

```
export OMP_NUM_THREADS=4
./a.out
```

???

- Runtime function: **omp_set_num_threads(n)**

```
 omp_set_num_threads(4);
!$omp parallel
   :
```

???

- OMP PARALLEL clause: **num_threads(n)**

```
!$omp parallel num_threads(4)
```

# Useful OpenMP Functions

- Runtime function **omp_get_num_threads()**
  - ✓ Returns number of threads in parallel region
  - ✓ Returns 1 if called outside parallel region
- Runtime function **omp_get_thread_num()**
  - ✓ Returns id of thread in team
  - ✓ Value between [0,n-1] // where n = #threads
  - ✓ Master thread always has id 0
- Runtime function **omp_get_max_threads()**
  - ✓ Returns upper bound #threads in parallel region
  - ✓ value of OMP_NUM_THREADS or set_num_threads

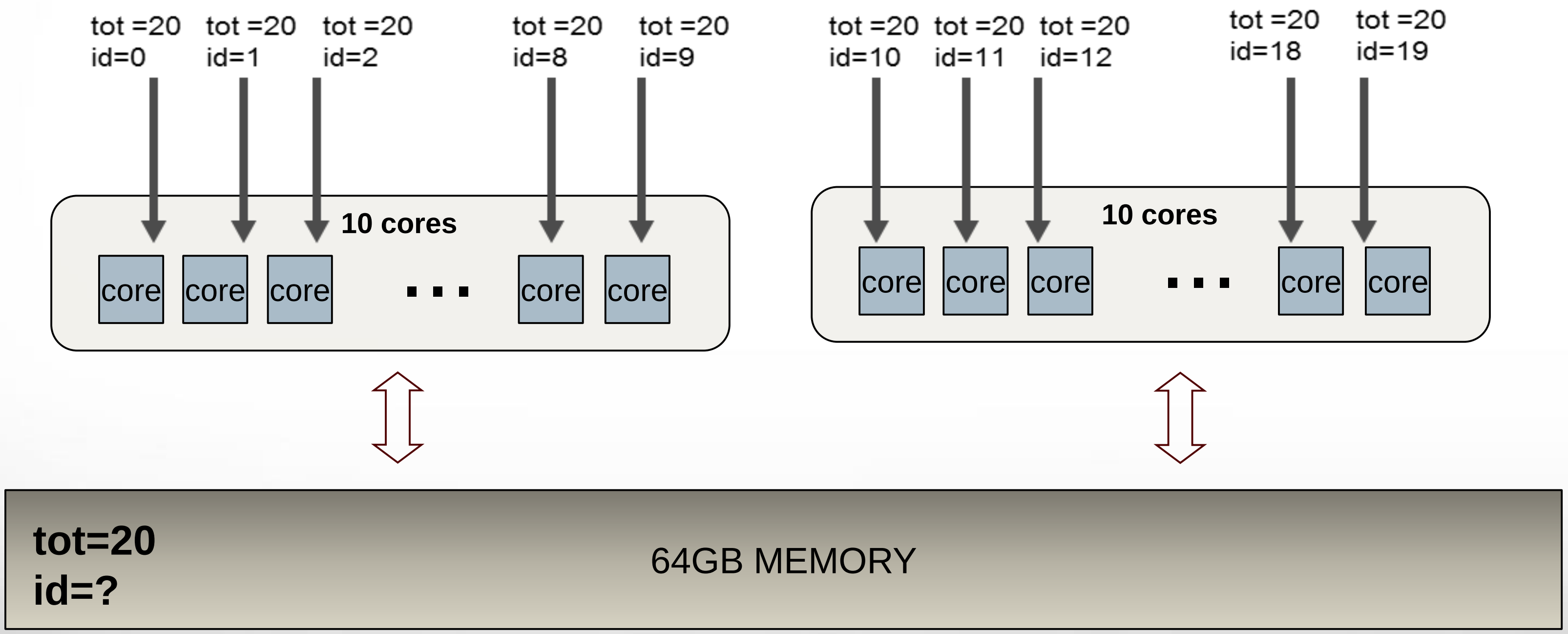
## DEMO 1

We will create an OpenMP program that does the following:
1) print total number of threads
2) start parallel region
3) every thread prints it's id and total number of threads
4) close the parallel region

NOTE: demo hello_threads

## Remember: memory is (conceptually) shared by all threads

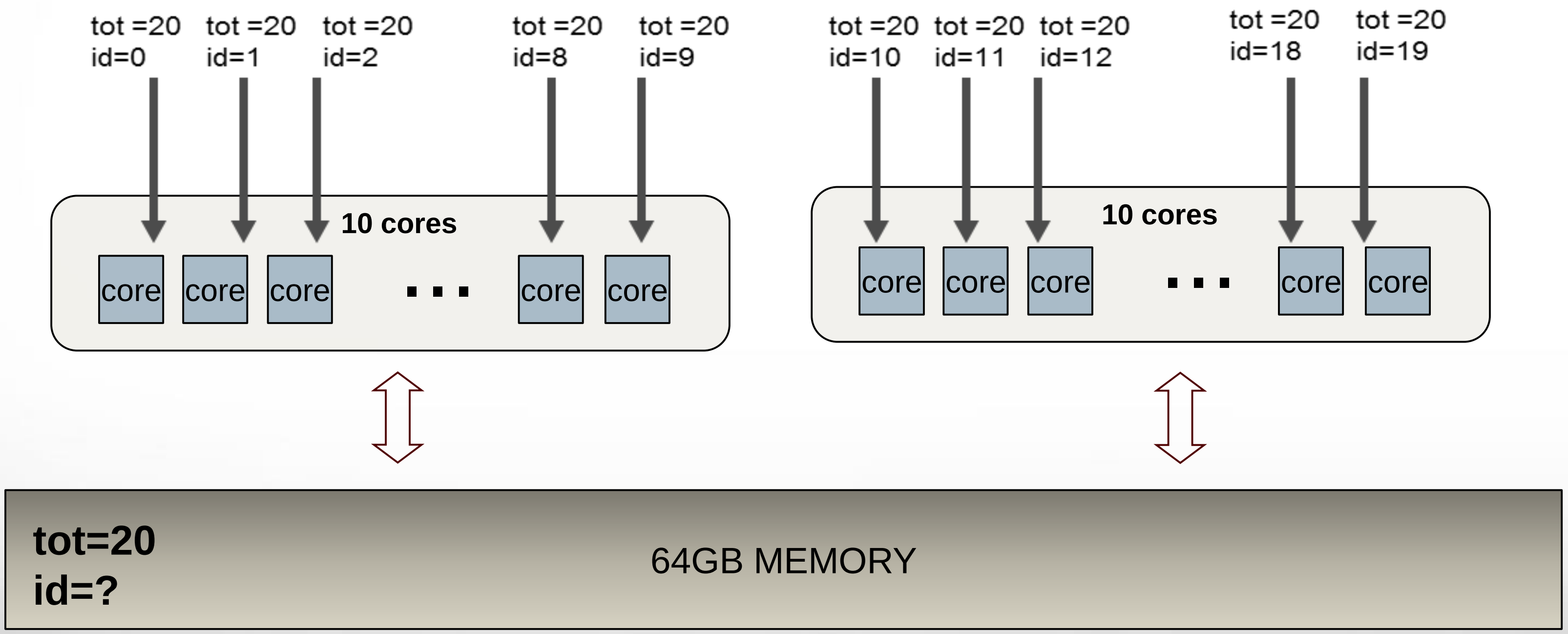
```
$OMP PARALLEL
TOT = omp_get_num_threads()
ID = omp_get_thread_id()
                :
!$OMP END PARALLEL
```



## Remember: memory is (conceptually) shared by all threads

```
$OMP PARALLEL
TOT = omp_get_num_threads()
ID = omp_get_thread_id()
              :
!$OMP END PARALLEL
```

***All threads try to access the same variable (possibly at the same time). This can lead to a race condition. Different runs of same program might give different results because of these race conditions***



## Data Scope Clauses

Data scope clauses: **private(list)**

```
!$OMP PARALLEL PRIVATE(a,c)
   :
!$OMP END PARALLEL
```

```
#pragma omp parallel private(a,c)
```

- Every thread will have it's own **"private"** copy of variables in list
- No other thread has access to this **"private"** copy
- Private variables are NOT initialized with value before region started.
- Private variables are NOT accessible after enclosing region finishes

***Index variables (Fortran, C/C++) and variables declared inside parallel region (C/C++) are considered private by default.***

## Data Scope Clauses

Data scope clauses: **shared(list)**

```
!$OMP PARALLEL SHARED(a,c)
   :
!$OMP END PARALLEL
```

```
#pragma omp parallel shared(a,c)
```

- All variables in list will be considered shared.
- Every openmp thread has access to all these variables
- Programmer's responsibility to avoid race conditions

***By default most variables in work sharing constructs are considered shared in OpenMP. Exceptions include index variables (Fortran, C/C++) and variables declared inside parallel region (C/C++).***

## Data Scope Clauses

### Data scope clauses: **firstprivate(list)**

```
!$OMP PARALLEL FIRSTPRIVATE(a,c)
!$OMP END PARALLEL
```

```
#pragma omp parallel firstprivate(a,c)
```

- Very similar to private clause
- Private copies are initialized with value of original variable

### Data scope clauses: **default(** *shared* | *private* | *firstprivate* | *lastprivate* **)**

```
!$OMP PARALLEL DEFAULT(private)
!$OMP END PARALLEL
```

```
#pragma omp parallel default(private)
```

- Sets default data scoping rule
- If not set, default depends on directive
    - e.g. shared for work sharing directives

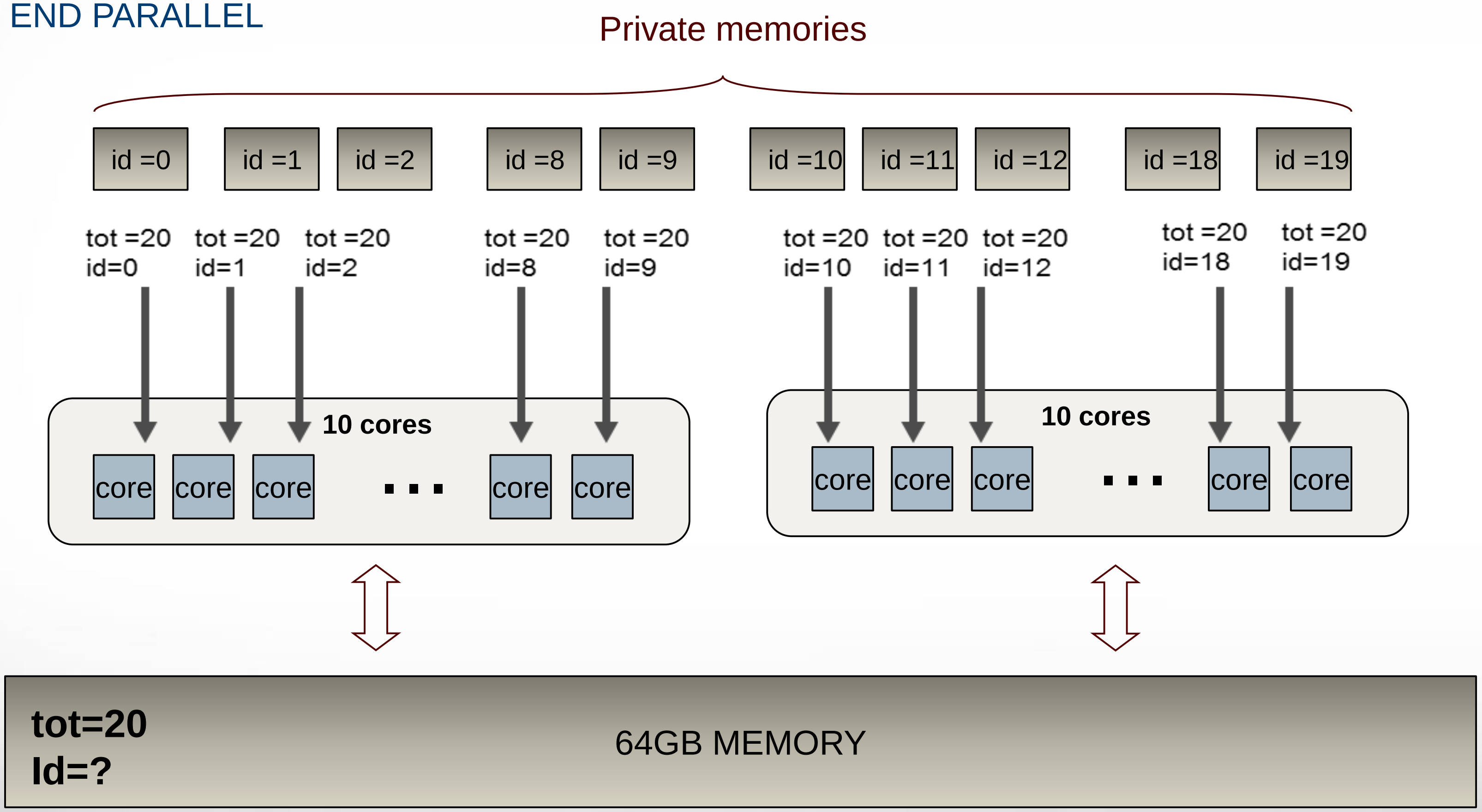
## DEMO 2

We will rewrite the OpenMP program (from previous demo) and make sure all variables are assigned and printed correctly.

NOTE: exercise hello_threads

## Remember: memory is (conceptually) shared by all threads

```
$OMP PARALLEL PRIVATE(ID)
TOT = omp_get_num_threads()
ID = omp_get_thread_id()
               :
!$OMP END PARALLEL
```



## TIP 1: Stack size

- OpenMP creates separate data stack for every worker thread to store copies of private variables (master thread uses regular stack)
- Size of these stacks is not defined by OpenMP standards
- Behavior of program undefined when stack space exceeded
  - ✓ Although most compilers/RT will throw seg fault
- To increase stack size use environment var OMP_STACKSIZE, e.g.
  - ✓ export OMP_STACKSIZE=512M
  - ✓ export OMP_STACKSIZE=1G
- To make sure master thread has large enough stack space use ulimit -s command (unix/linux).

## Work Sharing Directives (FORTRAN)

Work sharing directive (Fortran): **!$OMP DO** *[clauses]*

```
      :
!$OMP PARALLEL
!$OMP DO
DO n=1,N
   A(n) = A(n) + B
ENDDO
!$OMP END DO
!$OMP END PARALLEL
      :
```

**OR**

```
      :
!$OMP PARALLEL  DO
DO n=1,N
   A(n) = A(n) + B
ENDDO
!$OMP END PARALLEL DO
      :
```

- DO command must immediately follow “!$OMP DO” directive
- Loop iteration variable is “private” by default
- If “end do” directive omitted it is assumed at end of loop
- Not case sensitive

## Work Sharing Directives (C/C++)

Work sharing pragma (C/C++): **#pragma omp for** *[clauses]*

```
   :
#pragma omp parallel
#pragma omp for
   for (int i=1;i<N;++i)
      A(n) = A(n) + B;

   :
```

**OR**

```
   :
#pragma omp parallel for
   for (int i=1;i<N;++i)
      A(n) = A(n) + B;

   :
```

- ***for*** command must immediately follow "***#pragma omp for***"
- Newline required after "***#pragma omp for***"
- Iteration variable can be
  - ✓ Signed/unsigned integer variable
  - ✓ Pointer type
  - ✓ Random access iterator

## Work Sharing Directives (C/C++)

**C++**

***new in OpenMP 3.0.***

**Random access iterators:**

```
vector<int> vec(10);
vector<int>::iterator it= vec.begin();
#pragma omp parallel for
  for ( ; it != vec.end() ; ++it) {
      // do something with *it
  }
```

**Pointer type:**

```
int N = 1000000;
int arr[N];
#pragma  omp parallel for
  for (int* t=arr;t<arr+N;++t) {
    // do something with *t
}
```

## DEMO 3

We will create a program that computes a simple matrix vector multiplication b=Ax, either in fortran or C/C++. We will use OpenMP directives (pragmas) to make it run in parallel.

NOTE: demo matrix_mult

## Data Dependencies

Not all loops can be parallelized. Before adding OpenMP directives we need to check for **data dependencies**

For example:

```
for (i=1 ; i<N ; ++i)
   A[i] = A[i-1] + 1
end
```

**Is the result guranteed to be correct if you run this loop in parallel?**

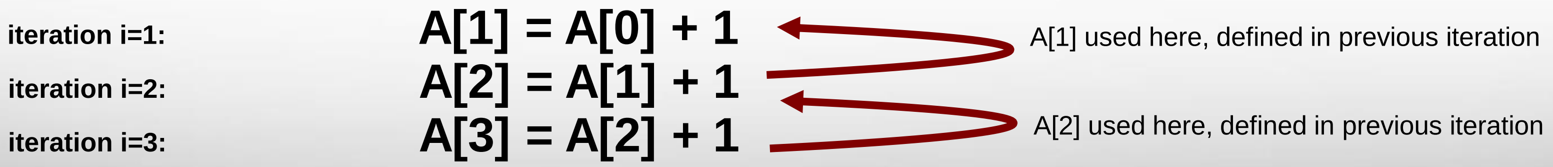
## Data Dependencies

Not all loops can be parallelized. Before adding OpenMP directives we need to check for **data dependencies**

For example:

```
for (i=1 ; i<N ; ++i)
   A[i] = A[i-1] + 1
end
```

Easier to see when you unroll loop (partly):



**iteration i=1:** A[1] = A[0] + 1
**iteration i=2:** A[2] = A[1] + 1
**iteration i=3:** A[3] = A[2] + 1

A[1] used here, defined in previous iteration
A[2] used here, defined in previous iteration

# REDUCTION

Data scope clause: **REDUCTION(*op:list*)**

In a reduction the values of the local copies will be summarized (reduced) into a global shared variable (using the specified reduction operator).

Example:

```
for (int i=0;i<10;++i)
  sum=sum+a[i];
```

```
#pragma omp parallel for reduction(+:sum)
for (int i=0;i<10;++i)
    sum=sum+a[i];
```

- ➢ Only certain kind of operators allowed
  - ✓ +, - , * , max, min
  - ✓ & , | , ^ , && , || (C++)
  - ✓ .and. , .or. , .eqv. , .neqv. , iand , ior , ieor (Fortran)
  - ✓ OpenMP 4.0 allows for user defined reductions
- ➢ Variables in list have to be *shared*

Also allowed in c/c++ since OpenMP 3.1

## DEMO 4

We will create a subroutine that computes the dot product of two vectors. We will use OpenMP pragmas (directives) to make it run in parallel.

NOTE: demo dot_product

# TIP 2: ORPHANED DIRECTIVES

An OpenMP directive (pragma) that appears independently from another enclosing directive is called an orphaned directive (pragma). It exists outside of another directive's (pragma) static (lexical) extent. For example:

```
int main() {
#pragma omp parallel
   foo()
   return 0;
}
```

```
void foo() {
#pragma omp for
   for (int i=0;i<N;i++) {….}
}
```

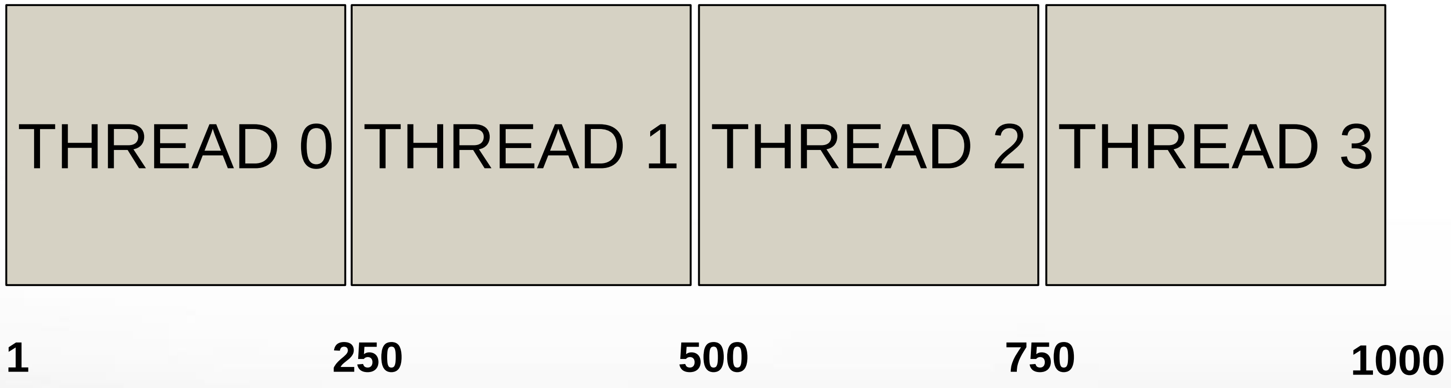
***Note: OpenMP directives (pragmas) should be in the dynamic extent of a parallel section directive (pragma).***

## Scheduling Clauses

**SCHEDULE (STATIC,250)** **//loop with 1000 iterations, 4 threads**

```
!$OMP PARALLEL DO SCHEDULE (STATIC,250)
DO i=1,1000
   :
ENDDO
!$OMP END PARALLEL DO
```

```
#pragma omp parallel for schedule(static,250)
 for (int i=0;i<1000;++i) {
         :
```



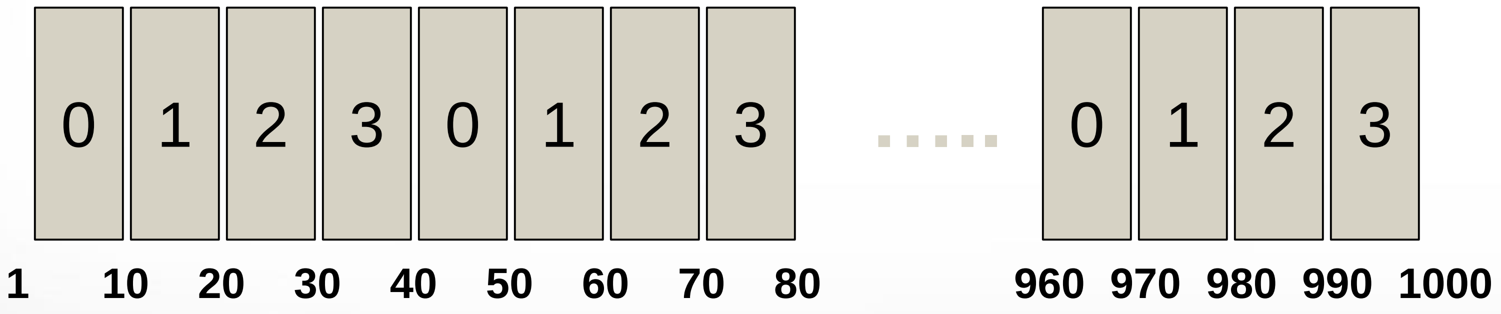
Although the OpenMP standard does not specify how a loop should be partitioned most compilers split the loop in N/p (N #iterations, p #threads) chunks by default.

# Scheduling Clauses

**SCHEDULE (STATIC,10)** **//loop with 1000 iterations, 4 threads**

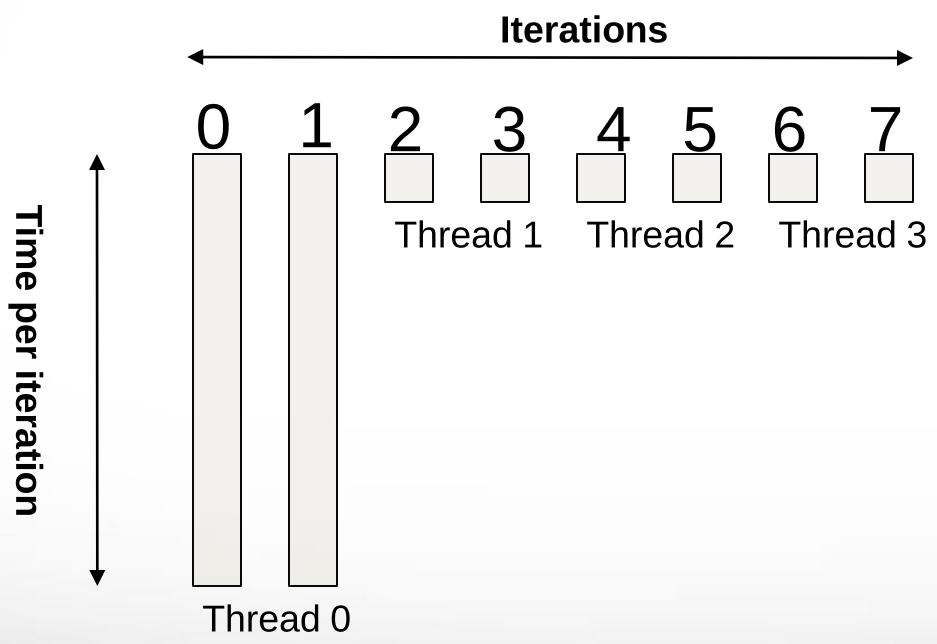
```
!$OMP PARALLEL DO SCHEDULE (STATIC,10)
DO i=1,1000
   :
ENDDO
!$OMP END PARALLEL DO
```

```
#pragma omp parallel for schedule(static,10)
   for (int i=0;i<1000;++i) {
            :
    }
```



## Scheduling Clauses

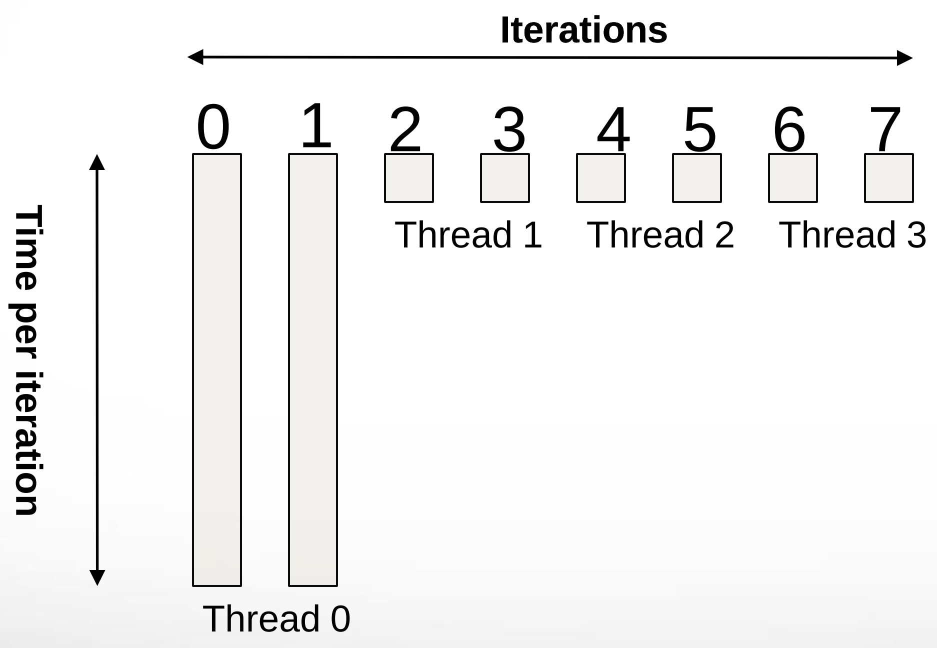
With static scheduling the number of iterations is evenly distributed among all openmp threads (i.e. Every thread will be assigned similar number of iterations). This is not always the best way to partition. Why is This?



***How can this happen?***

## Scheduling Clauses

With static scheduling the number of iterations is evenly distributed among all openmp threads (i.e. Every thread will be assigned similar number of iterations). This is not always the best way to partition. Why is This?



***This is called load imbalance. In this case threads 2,3, and 4 will be waiting very long for thread 1 to finish***

***<u>How can this happen?</u>***

## Scheduling Clauses

**SCHEDULE (DYNAMIC,10)** **//loop with 1000 iterations, 4 threads**

```
!$OMP PARALLEL DO SCHEDULE (DYNAMIC,10)
DO i=1,1000
   :
ENDDO
!$OMP END PARALLEL DO
```

```
#pragma omp parallel for schedule(dynamic,10)
 for (int i=0;i<1000;++i) {
              :
      }
```

Loop iterations are divided into pieces of size chunk. When  a thread finishes one chunk, it is dynamically assigned another.

***NOTE: there is a significant overhead involved compared to static scheduling. WHY?***

## Scheduling Clauses

**SCHEDULE (GUIDED,10)** **//loop with 1000 iterations, 4 threads**

```
!$OMP PARALLEL DO SCHEDULE (GUIDED,10)
DO i=1,1000
   :
ENDDO
!$OMP END PARALLEL DO
```

```
#pragma omp parallel for schedule(guided,10)
   for (int i=0;i<1000;++i) {
                :
       }
```

Similar to DYNAMIC schedule except that chunk size is relative to number of iterations left.

***NOTE: there is a significant overhead involved compared to static scheduling. WHY?***

## Work Sharing Directives (2)

**#pragma omp single (!$OMP SINGLE)**

Although technically it doesn't share work, another work sharing pragma that OpenMP provides is **#pragma omp single (!$OMP SINGLE)** When encountering a single directive only one member of the team will execute the code in the block

- One thread (not neccesarily master) executes the block
- Other threads will wait
- Useful for thread-unsafe code
- Useful for I/O operations

## Work Sharing Directives (3)

**#pragma omp sections (!$OMP SECTIONS)**

Executes set of structured blocks in parallel

```
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
   // WORK 1
#pragma omp section
   // WORK 2
}
```

```
!$OMP PARALLEL
!$OMP SECTIONS

!$OMP SECTION
      // WORK 1
!$OMP SECTION
      // WORK 2
!$OMP END SECTIONS
!$OMP END PARALLEL
```

This will execute ”WORK 1” and ”WORK 2” in parallel

# NOWAIT Clause

Worksharing constructs have an implicit barrier at the end of their worksharing region. However, OpenMP provides a clause to ommit this barrier.

```
#pragma omp for nowait
    :
```

```
!$OMP DO
    :
!$OMP END DO NOWAIT
```

- At end of work sharing constructs threads will not wait
- There is always barrier at end of parallel region

# Work Sharing Summary

## OMP work sharing constructions discussed

| | |
|---|---|
| ➢ #pragma omp for | (!$OMP DO) |
| ➢ #pragma omp sections | (!$OMP SECTIONS) |
| ➢ #pragma omp single | (!$OMP SINGLE) |

## Clauses that can be used with these constructs (

incomplete list and not all clauses can be used with every directive)

- SHARED (list)
- PRIVATE (list)
- FIRSTPRIVATE (list)
- LASTPRIVATE(list)
- SCHEDULE (STATIC | DYNAMIC | GUIDED, chunk)
- REDUCTION(op:list)
- NOWAIT

## Communication/Synchronization

OpenMP programs use shared variables to communicate. We need to make sure these variables are not accessed at the same time by different threads (will cause race conditions). OpenMP provides a number of directives for synchronization.

## Synchronization Directives (1)

**#pragma omp master (!$OMP MASTER)**

This Directive ensures that only the master threads excecutes instructions in the block. There is no implicit barrier so other threads will not wait for master to finish

***What is difference with !$OMP SINGLE DIRECTIVE?***

## Synchronization Directives (2)

**#pragma omp critical (!$OMP CRITICAL)**

This Directive makes sure that only one thread can execute the code in the block. If another threads reaches the critical section it will wait untill the current thread finishes this critical section. Every thread will execute the critical block and they will synchronize at end of critical section

- Introduces overhead
- Serializes critical block
- If time in critical block relatively large → speedup negliable

## DEMO 5

In the REDUCTION exercise we created an OpenMP program that computes the dotproduct of two vectors using the reduction clause. Now we will create a version that uses critical blocks instead.

NOTE: dempo dotproduct_critical

## Synchronization Directives (3)

### #pragma omp atomic (!$OMP ATOMIC)

This Directive is very similar to the !$OMP CRITICAL directive on the previous slide. Difference is that !$OMP ATOMIC is only used for the update of a memory location. Sometimes !$OMP ATOMIC is also refered to as a mini critical section.

- Block consists of only one statement
- Atomic statement must follow specific syntax

## Synchronization Directives (4)

**#pragma omp barrier (!$OMP BARRIER)**

A barrier will force every thread to wait at the barrier until all threads have reached the barrier. The following omp directives we discussed before include an implicit barrier:

- !$ OMP END PARALLEL
- !$ OMP END DO
- !$ OMP END SECTIONS
- !$ OMP END SINGLE
- !$ OMP END CRITICAL

## TIP 3: IF Clause

OpenMP provides another useful clause to decide at run time if a parallel region should actually be run in parallel (multiple threads) or just by the master thread:

```
IF (logical expr)
```

For example:

```
$!OMP PARALLEL IF(n > 100000)          (fortran)
#pragma omp parallel if (n>100000)      (C/C++)
```

This will only run the parallel region when n> 100000

# Nested Parallelism

## Nested Parallelism

OpenMP allows parallel regions inside other parallel regions ,e.g.

```
#pragma omp parallel for
for (int i=0; i<N;++i) {
:
#pragma omp parallel for
for (j=0;j<M;++j)
}
```

- ➢ To enable nested parallelism:
  - ✓ env var: OMP_NESTED=1
  - ✓ lib function: omp_set_nested(1)
- ➢ To specify number of threads:
  - ✓ omp_set_ num_threads()
  - ✓ OMP_NUM_THREADS=4,2

*NOTE: using nested parallelism does introduce extra overhead and there is a possibility of over-subscription of threads*

## MKL

The Intel Math Kernel Library (MKL) has very specialized and optimized versions of many math functions (e.g. blas, lapack). Many of these have been parallelized using OpenMP.

To run mkl functions in parallel use:

- MKL_NUM_THREADS
- OMP_NUM_THREADS

For more information about MKL:

http://hprc.tamu.edu/wiki/index.php/Ada:MKL