



# Introduction to OpenMP

Marinus Pennings

November 2, 2018



**DIVISION OF RESEARCH**  
TEXAS A & M UNIVERSITY

# Agenda

- What is openMP?
- Starting parallel region
- Data Scopes
- Work sharing
- Dependencies and Reductions
- Synchronization
- Scheduling
- OpenMP tasks

## Short course home page:

[https://hprc.tamu.edu/training/intro\\_openmp.html](https://hprc.tamu.edu/training/intro_openmp.html)

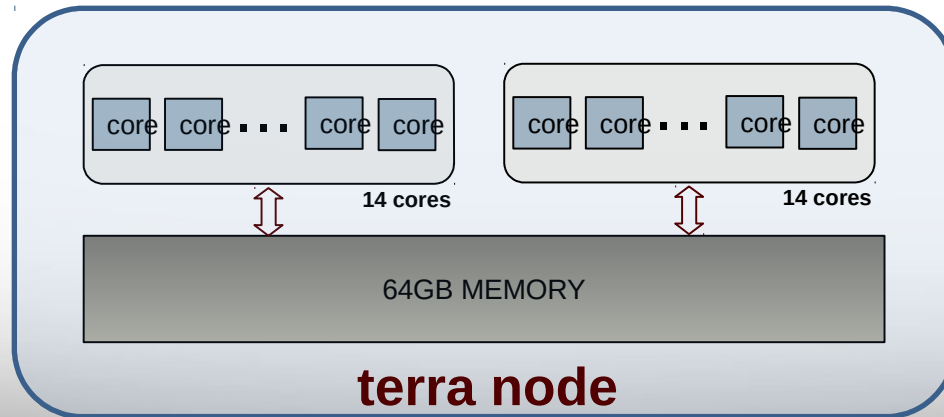
## Setting up OpenMP sample codes:

- On ada/curie type: `/scratch/training/OpenMP/setup.sh`
- On terra type: `/scratch/training/OpenMP/setup.sh`

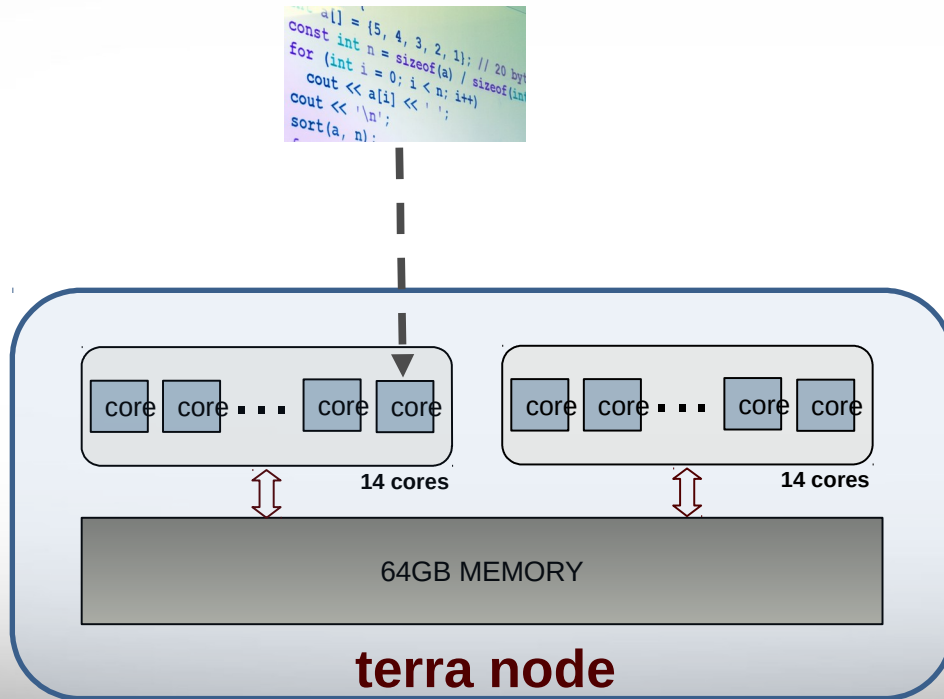
# Basic Computer Architecture

Each terra **NODE** has 28 cores (two 14 core cpus) per node and at least 64GB of **SHARED** memory (NOTE: ada has 20 cores per node and curie has 16)

All modern computers have multiple processing cores. Average desktop has around 4 cores. On our HPRC cluster terra, each **NODE** has 28 cores (two 14 core cpus) per node and at least 64GB of **SHARED** memory (NOTE: ada has 20 cores per node and curie has 16)

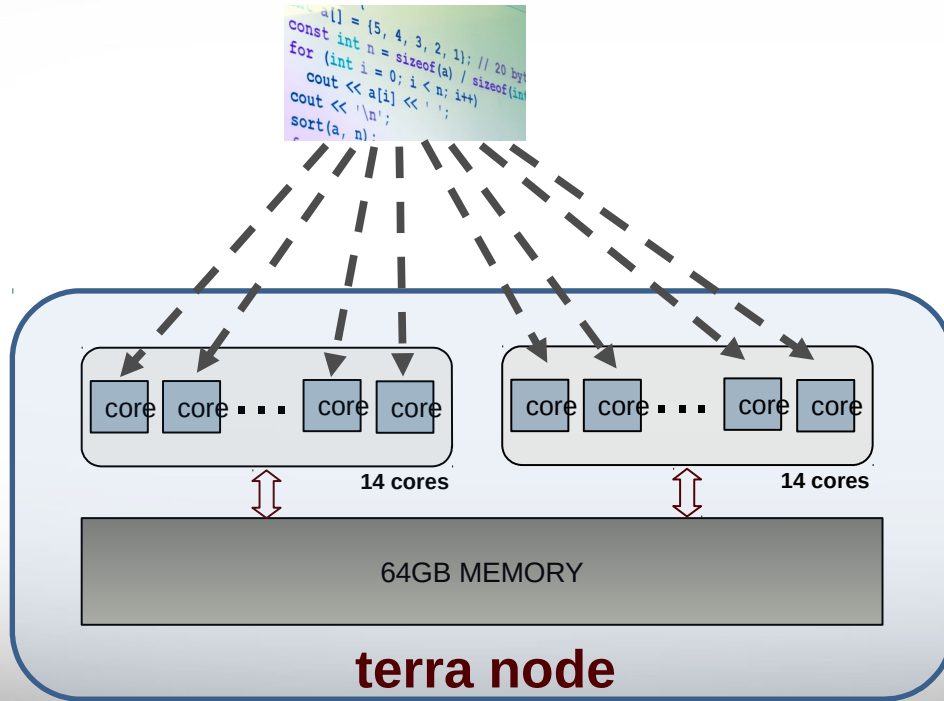


# Basic Computer Architecture



All modern computers have multiple processing cores. Average desktop has around 4 cores. On our HPRC cluster terra, each **NODE** has 28 cores (two 14 core cpus) per node and at least 64GB of **SHARED** memory (NOTE: ada has 20 cores per node and curie has 16)

# Basic Computer Architecture



All modern computers have multiple processing cores. Average desktop has around 4 cores. On our HPRC cluster terra, each **NODE** has 28 cores (two 14 core cpus) per node and at least 64GB of **SHARED** memory (NOTE: ada has 20 cores per node and curie has 16)

# What is OpenMP?

De facto 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

*In a nutshell: using OpenMP, you can make a serial program run in parallel by annotating parts of the code that you want to run in parallel*


## C/C++ pragma format:

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

 New line required

## fortran directive format:

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

 Not case sensitive

# Starting Parallel Region

```
// some C/C++ code  
#pragma omp parallel  
{  
    // code block, will be  
    // executed in parallel  
}  
  
// more C/C++ code
```

```
c some fortran code  
!$OMP PARALLEL  
c code block, will be  
c executed in parallel  
!$OMP END PARALLEL  
  
c more fortran code
```

This will start an OpenMP region. A team of threads will be created, the code inside the parallel block will be executed concurrently by all threads.



# HelloWorld

SOURCE

## Exercise:

- 1) Create OpenMP version of HelloWorld ( either C/C++ or Fortran)
  - a) Create parallel region
  - b) Every thread prints Hello World
  - c) Close the parallel region
- 2) Compile the program (you can use GNU or Intel compiler)
- 3) Execute the program

# HelloWorld

## SOURCE

```
#include <iostream>
```

```
int main() {
```

```
#pragma omp parallel
```

```
{
```

```
    std::cout << "Hello World\n";
```

```
}
```

```
return 0;
```

```
}
```

pragma



```
program HELLO
```

```
!$OMP PARALLEL
```

```
print *, "Hello World"
```

```
!$OMP END PARALLEL
```

```
end program HELLO
```

directive



## COMPILING

**Need to include flag to tell the compiler to process the OpenMP pragmas/directives**

```
intel: icpc -qopenmp -o hi.x hello.cpp
```

```
gnu: g++ -fopenmp -o hi.x hello.cpp
```

```
intel: ifort -qopenmp -o hi.x hello.f90
```

```
gnu: gfortran -fopenmp -o hi.x
```

```
hello.f90
```

**Compile the program , and run again**

# HelloWorld

## SOURCE

```
#include <iostream>
```

```
int main() {
```

```
#pragma omp parallel
```

```
{
```

```
    std::cout << "Hello World\n";
```

```
}
```

```
return 0;
```

```
}
```

pragma



```
program HELLO
```

```
!$OMP PARALLEL
```

```
print *, "Hello World"
```

```
!$OMP END PARALLEL
```

```
end program HELLO
```

directive



## COMPILING

**Need to include flag to tell the compiler to process the OpenMP pragmas/directives**

```
intel: icpc -qopenmp -o hi.x hello.cpp
```

```
gnu: g++ -fopenmp -o hi.x hello.cpp
```

```
intel: ifort -qopenmp -o hi.x hello.f90
```

```
gnu: gfortran -fopenmp -o hi.x
```

```
hello.f90
```

## RUNNING

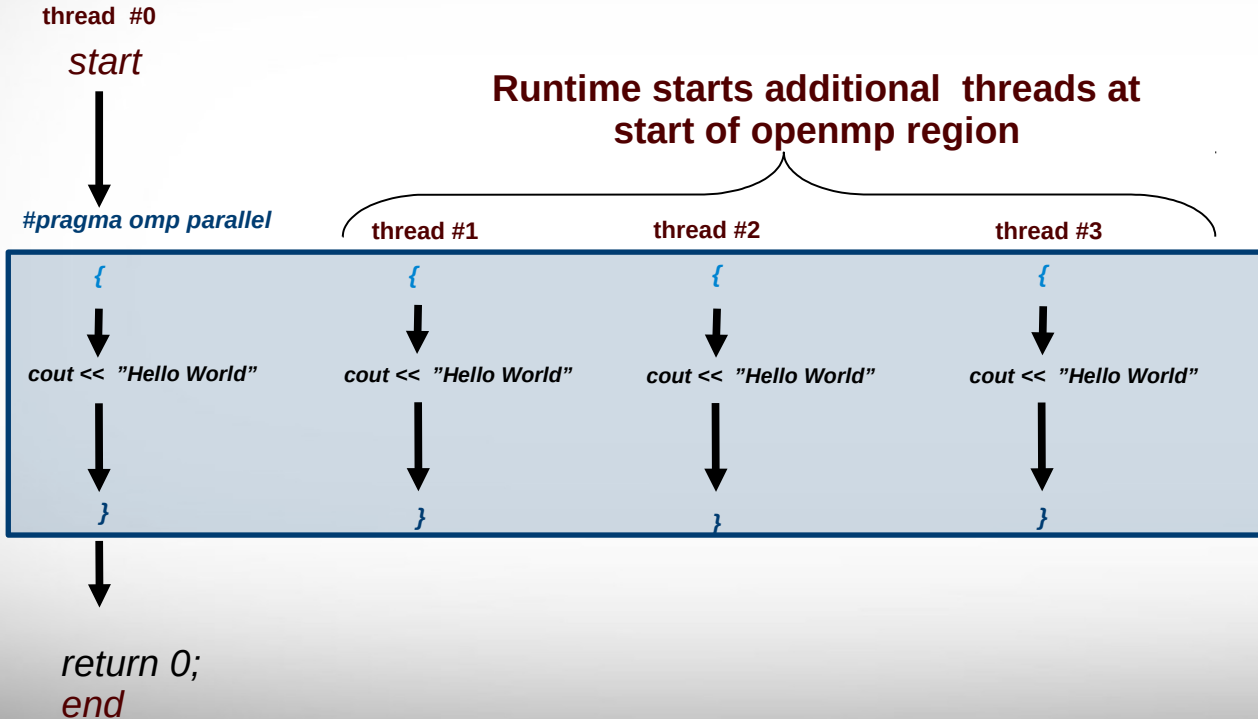
```
export OMP_NUM_THREADS=4
```

```
./hi.x
```

**Run the program again**

( I promise, it will work now)

# Fork/Join



```
#include <iostream>  
#include <omp.h>  
  
using std;  
int main() {  
  #pragma omp parallel  
  {  
    cout << "Hello world\n";  
  }  
  return 0;  
}
```

# Threads & Cores

**(OpenMP) THREAD:** Independent sequence of code, with a single entry and a single exit

**CORE:** Physical processing unit that receives instructions and performs calculations, or actions, based on those instructions.

- 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.

# Getting Thread info

- Runtime function: **omp\_get\_thread\_num()**

```
id = omp_get_thread_num(); // 0
#pragma omp parallel
{
    id = omp_get_thread_num(); // <thread id in region>
}
```

- Runtime function: **omp\_get\_num\_threads()**

```
tot = omp_get_num_threads(); // 1
#pragma omp parallel
{
    tot = omp_get_num_threads(); // < total #threads in region>
}
```

# Setting the number of Threads

- Environmental variable: **OMP\_NUM\_THREADS**

case sensitive



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

- Runtime function: **omp\_set\_num\_threads(n)**

???



```
omp_set_num_threads(4);  
#pragma omp parallel  
:  
:
```

- OMP PARALLEL clause: **num\_threads(n)**

???



```
#pragma omp parallel num_threads(4)
```

# Hello Threads

## Exercise:

- 1) Create OpenMP HelloThreads program that does the following:
  - a) Create parallel region
  - b) Every thread prints its own thread id and the total number of threads
  - c) Close the parallel region
- 2) Compile the program (you can use GNU or Intel compiler)
- 3) Execute the program

HINT: since you will be using OpenMP library functions you will want to include:

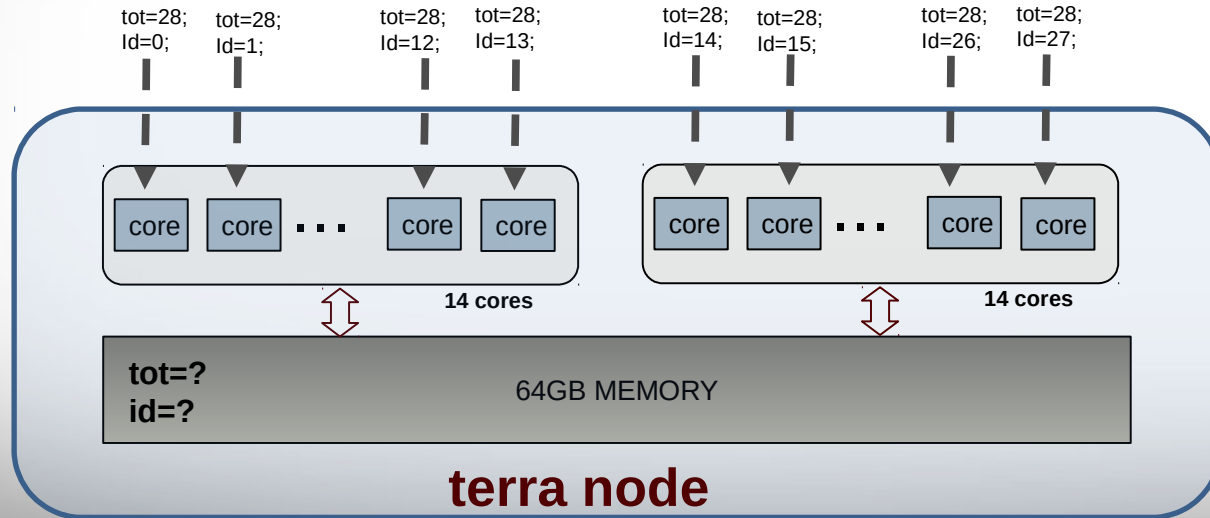
C/C++ : *#include "omp.h"*

Fortran : *use omp\_lib*



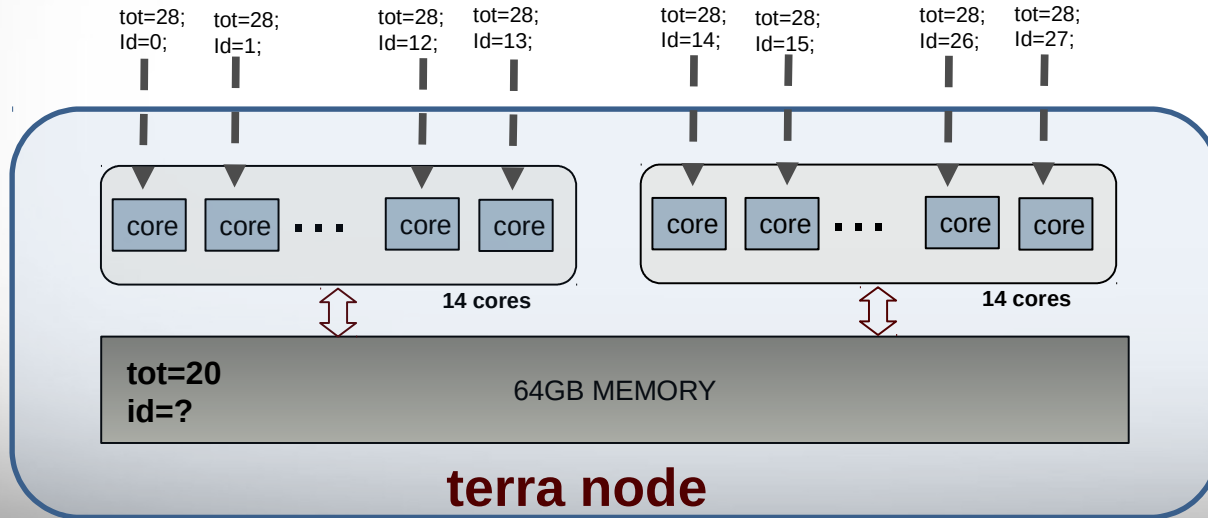
```
#pragma omp parallel
{
  tot = omp_get_num_threads();
  id = omp_get_thread_num();
}
```

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



```
#pragma omp parallel
{
  tot = omp_get_num_threads();
  id = omp_get_thread_num();
}
```

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



***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)**

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

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

- Every thread will have its 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 (use **firstprivate** instead)
- 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)**

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

```

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

```

- 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++).*

# Other Data Scope Clauses

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

- Every thread will have its own "**private**" copy of variables in list.
- No other thread has access to this "**private**" copy.
- **firstprivate** variables are initialized to value before region started.
- **firstprivate** variables are NOT accessible after end of enclosing region.

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

- Set default data scoping rule.
- If not set, default depends on the pragma/directive (e.g. Shared for "for" pragma).

demo datascope



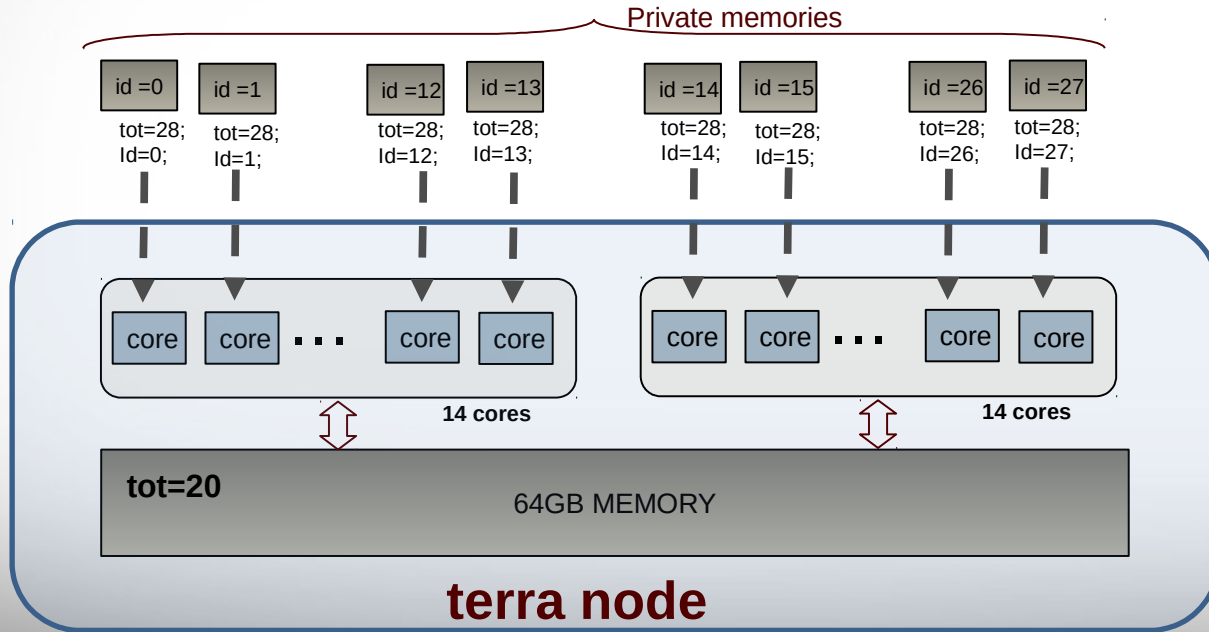
# Hello Threads (take 2)

## Exercise:

- 1) Create OpenMP HelloThreads program that does the following:
  - a) Create parallel region
  - b) Every thread prints **its own** thread id and the **total number of threads**
  - c) Close the parallel region
- 2) Compile the program (you can use GNU or Intel compiler)
- 3) Execute the program

```
#pragma omp parallel
{
  tot = omp_get_num_threads();
  id = omp_get_thread_num();
}
```

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



# TIP: Stack size

- OpenMP creates separate data stack for every worker thread to store 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 set stack size use environment var OMP\_STACKSIZE:
  - ✓ export OMP\_STACKSIZE=512M
  - ✓ export OMP\_STACKSIZE=1G
- To make sure master thread has large enough stack space use ulimit -s command (unix/linux).

*Let's create a demo program where the threads fill up the stack space*



# Work Sharing Directives

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

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

OR

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

- **for** command must immediately follow “**#pragma omp for**”
- Newline required after “**#pragma omp for**”
- Originally iteration variable could only be signed/unsigned integer variable.

# Work Sharing Directives

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
```

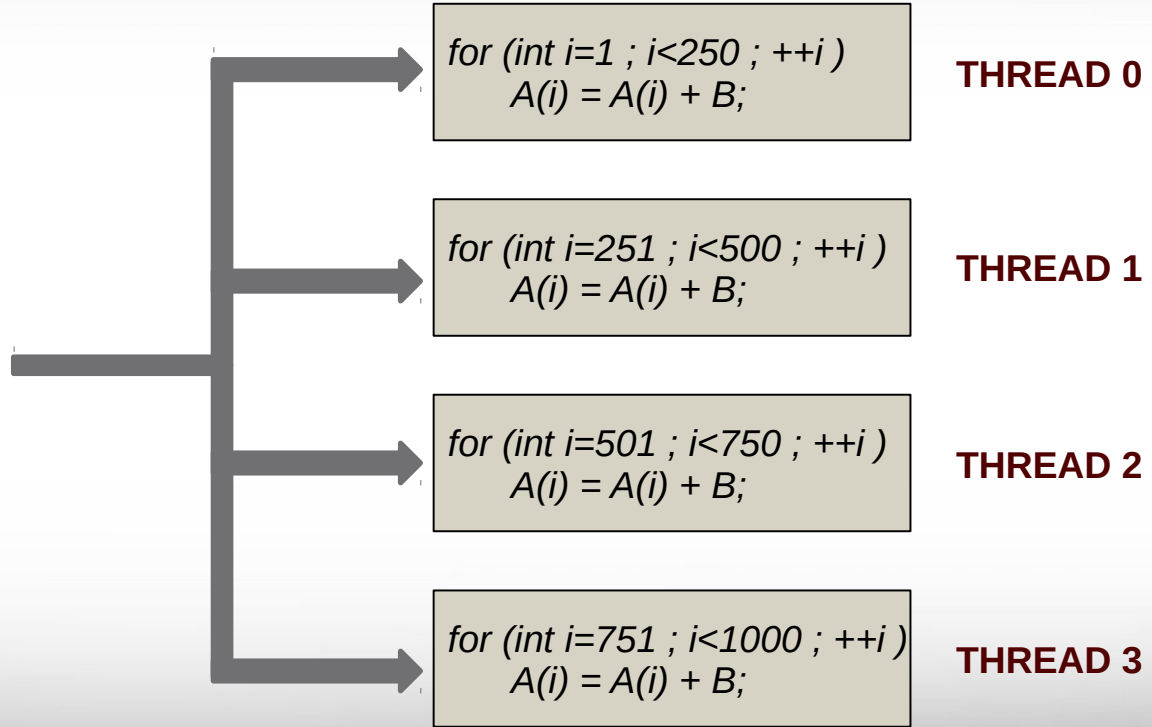
**OR**

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

```
#pragma omp parallel for  
for (int i=1;i<1000;++i)  
  A(i) = A(i) + B;
```



# Work Sharing Directives



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  
}
```

# Matrix multiplication

## Exercise:

- 1) Create program that computes simple matrix vector multiplication:
  - a) Use the OpenMP work sharing construct
  - b) Create as function that takes as arguments matrix and vector
  - c) Add timing to see the running time.
- 2) Compile and run the program
- 3) Try it with various number of threads
- 4) Vary the input size and see how it affects run time

# Manual Worksharing

## Case Study:

Assume the following dummy program:

```
A=int[N]; // set all elements to -1  
for (int i=0 ; i<N ; ++i) A[i] = i;
```

**We want to run this in parallel. Normally we would use the OpenMP worksharing directive. Let's do it without and partition the loop manually**

- 1) Start a parallel region.
- 2) Every thread will compute its offsets
- 3) Every thread will process part of the loop.

# TIP: ORPHANED PRAGMAS

An OpenMP pragma that appears independently from another enclosing pragma is called an orphaned pragma. It exists outside of another pragma static extent.

```
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).*

# Data Dependencies

Can all loops can be parallelized?

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



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

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



# Data Dependencies

Can all loops can be parallelized?

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



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

Unroll the loop (partly):

**iteration i=1:**

**iteration i=2:**

**iteration i=3:**

**A[1] = A[0] + 1**

**A[2] = A[1] + 1**

**A[3] = A[2] + 1**



A[1] used here, defined in previous iteration

A[2] used here, defined in previous iteration

# Reductions

*A reduction variable is a special variable that is **updated** during every iteration of a loop and there are no other definitions or uses of that variable. Update is always of the form “a = a op b”*

## Can we run this in parallel?

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



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

# Reductions

*A reduction variable is a special variable that is **updated** during every iteration of a loop and there are no other definitions or uses of that variable. Update is always of the form “a = a op b”*

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

- Only certain kind of operators allowed
  - ✓ +, -, \*, max, min,
  - ✓ &, |, ^, &&, || (C/ C++)
  - ✓ .and., .or., .eqv., .neqv., iand, ior, ieor (Fortran)
- OpenMP 4.0 allows for user defined reductions

*Reduction variable  
has to be shared*

```
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];
```

# Dot product (take 1)

## Exercise:

- 1) Create program that takes 2 vectors (arrays) and computes:
  - a) The dot product of the vectors
  - b) the largest element of the two vectors
- 2) Add timing to compute the run time
- 3) Compile and run the program
- 4) Try it with various number of threads

# User Defined Reductions

New in  
OpenMP  
4.0

```
#pragma omp declare reduction (name : type list : combiner) \  
initializer(initializer-expression)
```

```
!$ omp declare reduction (name : type list : combiner) \  
initializer(initializer-expression)
```

Example: UDR that computes sum (mimics + operator)

```
#pragma omp declare reduction (mysum : int : omp_out = omp_out + omp_in)  
initializer(omp_priv = 0)
```

*Fixed variable name  
to represent initializer*

*Fixed variable names to represent  
in and out of reduction*

# User Defined Reductions

## Case Study:

suppose we have a vector of random points (with x and y coordinates). We want to find the point with the longest distance ( $d = \sqrt{x^2 + y^2}$ )

- 1) Create a C++ class with
  - a) Two members: x and y coordinate
  - b) Member function that computes the distance
- 2) Create User Defined Reduction that takes pair of points and returns one with longest distance
- 3) Create OpenMP loop with reduction clause that computes the point with longest distance.
- 4) Compile and run with various number of threads.

# Work Sharing Directives

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

```
#pragma omp parallel
{
#pragma omp sections
{
#pragma omp section
    process(A1,A2)
#pragma omp section
    process(B1,B2)
}
}
```

- In an OpenMP sessions block, all "sections" will be executed concurrently
- Each section will be processed by a separate thread
- How is this different from **#pragma omp for**

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

```
#pragma omp parallel
{
#pragma omp single
{
    std::cout << "thread" <<

    omp_get_thread_num() <<
        " reached here first\n";
}
}
```

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

# NOWAIT Clause

Worksharing constructs have an implicit barrier at the end of their worksharing region. To omit 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

*NOTE: In example above the **nowait** clause is used with a **for/do** work sharing construct. It also works with the other worksharing construct: **sections** and **single***

demo nowait as part of single construct



# OpenMP Synchronization

OpenMP programs use shared variables to communicate. Need to make sure these variables are not accessed at the same time by different threads to avoid race conditions.

# Synchronization Directive

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

- **ALL** threads will execute the code inside the block
- Execution of the block is serialized, only one thread at a time will execute the block
- Threads will wait at start of block when another thread already inside the block

```
int tot=0; int id=0;
#pragma omp parallel
{
  #pragma omp critical
  {
    id = omp_get_thread_num(); tot=tot+id;
    std::cout << "id " << id << ", tot: " << tot << "\n";
  }
  // do some other stuff
}
```

Only one thread can execute  
block, other threads will wait

Will threads wait until all other threads  
have finished?

**NOTE:** If block consists of only a single assignment can use `#pragma omp atomic` instead

42

Thread 3 reaches critical block first, starts executing

Thread 1 reaches critical block. Thread 3 still executing, so has to wait

**Thread 3**

**Thread 1**

**Thread 0**

**Thread 2**

`#pragma omp critical`

```
{  
  // some code  
}
```

`#pragma omp critical`

```
{  
  // some code  
}
```

`#pragma omp critical`

```
{  
  // some code  
}
```

`#pragma omp critical`

```
{  
  // some code  
}
```

Thread 3 finished, will continue

Thread 1 finished, will continue

Thread 0 finished, will continue

Thread 1 finished, Thread 0 starts executing block

# Dot product (take 2)

## Exercise:

- 1) Create program that takes 2 vectors and computes:
  - a) The dot product of the vectors
  - b) the largest element of the two vectors
  - c) This time use OpenMP atomic blocks
- 2) Add timing to compute the run time
- 3) Compile and run the program
- 4) Try it with various number of threads

# Synchronization pragma

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

- **ONLY** master threads will execute the code inside the block
- Other threads will skip executing the block
- Other threads will not wait at end of the block

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

- **ALL** threads will wait at the barrier.
- Only when all threads have reached the barrier, each thread can continue
- Already seen implicit barriers, e.g. at the end of "#pragma omp parallel", "#pragma omp for"

# TIP: 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:

<code>\$!OMP PARALLEL IF(n &gt; 100000)</code>	(fortran)
<code>#pragma omp parallel if (n&gt;100000)</code>	(C/C++)

This will only run the parallel region when  $n > 100000$

# TIP: Printing OMP env vars

OpenMP 4.0 introduces a new environmental variable that instructs the runtime to print version number and all OpenMP environmental variables:

```
OMP_DISPLAY_ENV=VERBOSE
```

To only print the OpenMP version number, you can use:

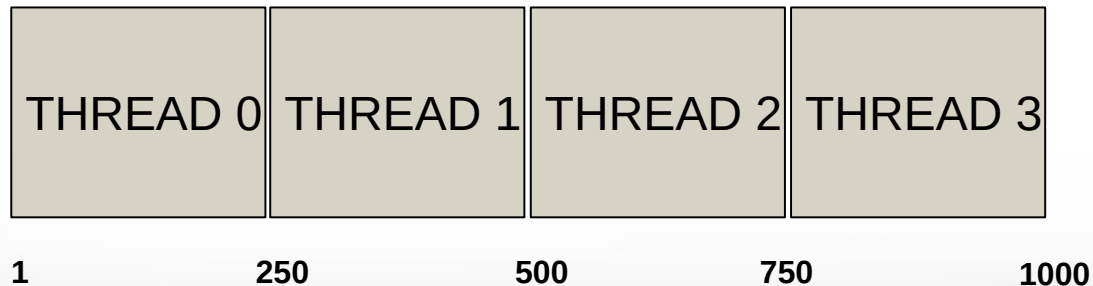
```
OMP_DISPLAY_ENV=TRUE
```

# 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.

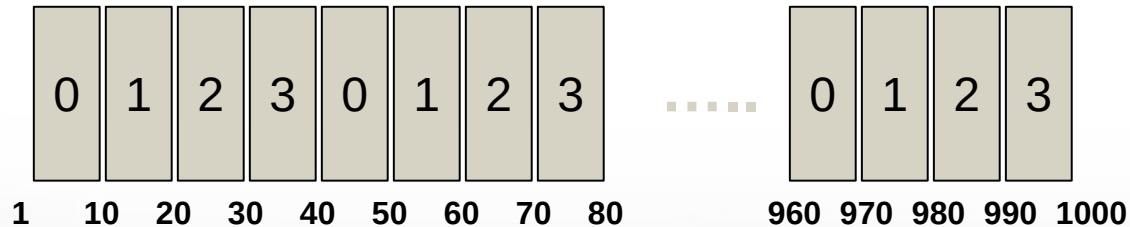


# Scheduling Clauses

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

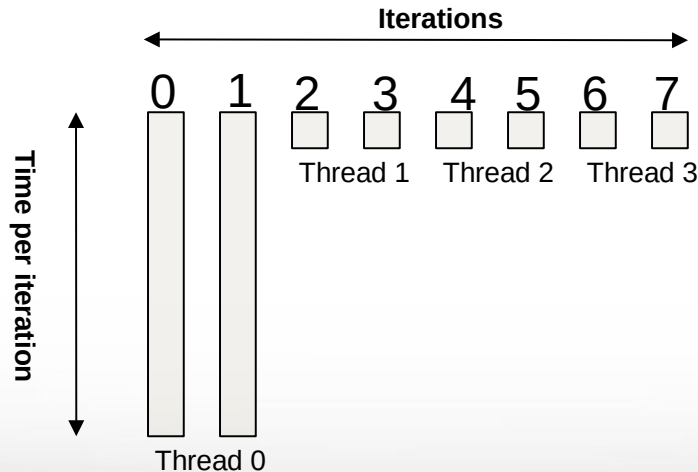
```
!$OMP PARALLEL DO SCHEDULE (STATIC,10)  
DO i=1,1000  
  :  
ENDDO  
!$OMP END PARALLEL DO
```

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



# Scheduling Clauses

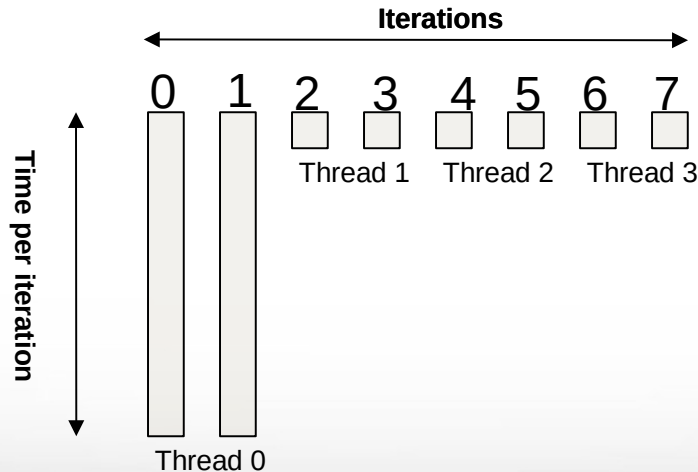
With static scheduling the number of iterations is evenly distributed among all openmp threads. This is not always the best way to partition. Why?



**How can this happen?**

# Scheduling Clauses

With static scheduling the number of iterations is evenly distributed among all openmp threads. This is not always the best way to partition. Why?



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

**How can this happen?**

# 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?***

# Nested Parallelism

OpenMP allows parallel regions inside other parallel regions

```
#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 might over-subscribe of threads*

# OpenMP Tasks

New in  
OpenMP  
3.0

*Especially useful for unbounded loops, Irregular algorithms,  
Tree/lists, Recursive algorithms, Producer/Consumer type problems.*

- Each task is independent unit of work
- When a thread encounter a task construct, thread decides to execute it itself or put in task pool
- Available threads will execute tasks
- Tasks consist of:
  - Code to execute
  - Data environment

Threads add tasks to pool

Available threads  
retrieve tasks from the  
pool and execute them

# OpenMP Tasks

## Creating tasks

### #pragma omp task

- Defines/creates new task
- Task will be added to task pool
- Idle thread will get tasks from pool and executes it
- Has to be in parallel region

## Synchronization

### #pragma omp taskwait

- Acts like a barrier
- Thread wait until all child tasks have finished

## Default Data Scope Rules

```
int b , c ;
#pragma omp parallel private ( b )
{
    int d ;
    #pragma omp task
    {
        int e;
        b , d; // firstprivate
        c;     // shared
        e;     // private
    }
}
```

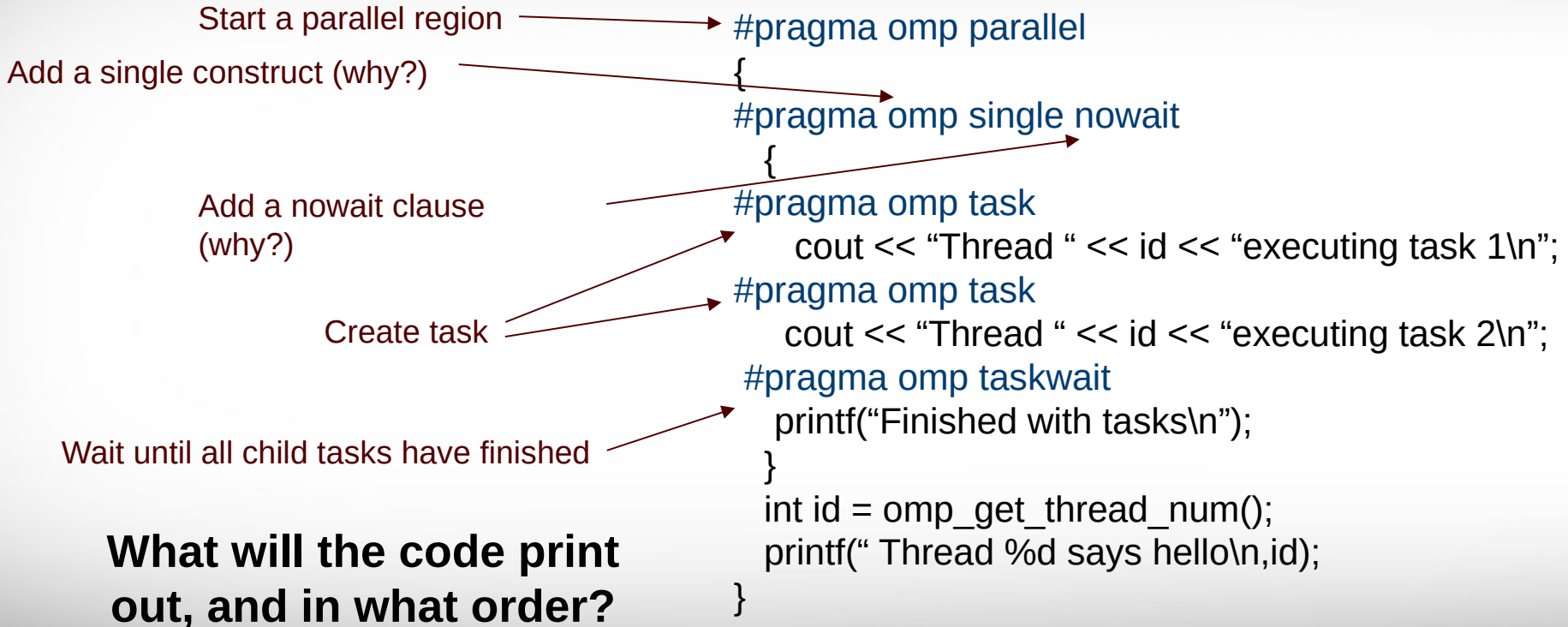
Enclosing data scope is private, inside task firstprivate

Enclosing data scope is shared inside task shared

Data scope is defined as private



# Analysis of OpenMP Tasks



# Dot product (take 3)

## Exercise:

- 1) Create program that takes 2 vectors and computes:
  - a) The dot product of the vectors
  - b) the largest element of the two vectors
  - c) This time, use OpenMP tasks
- 2) Add timing to compute the run time
- 3) Compile and run the program
- 4) Try it with various number 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.

- MKL\_NUM\_THREADS
- OMP\_NUM\_THREADS

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

# Questions?

You can always reach us at [help@hprc.tamu.edu](mailto:help@hprc.tamu.edu)

