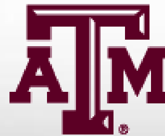
Introduction to OpenMP

Marinus Pennings

October 17,2017



Outline

- Basic Computer Architecture
- Starting parallel region
- Data Scopes
- Work sharing
- Dependencies and Reductions
- Bonus: Synchronization

Short course home page:

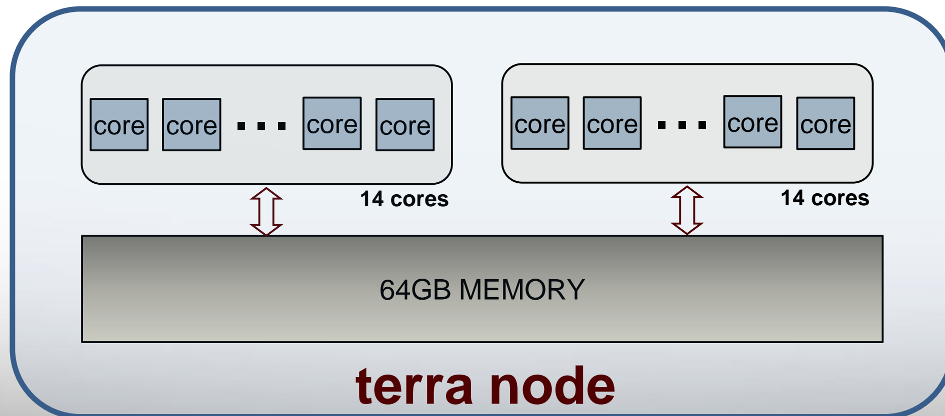
https://hprc.tamu.edu/training/intro_openmp.html

Setting up OpenMP sample codes:

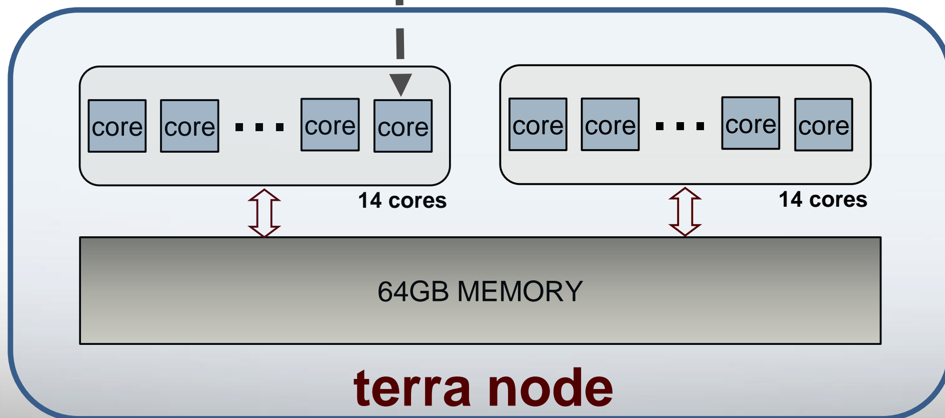
- On ada/curie type: /scratch/training/OpenMP/setup.sh
- On terra type: /scratch/training/OpenMP/setup.sh

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)

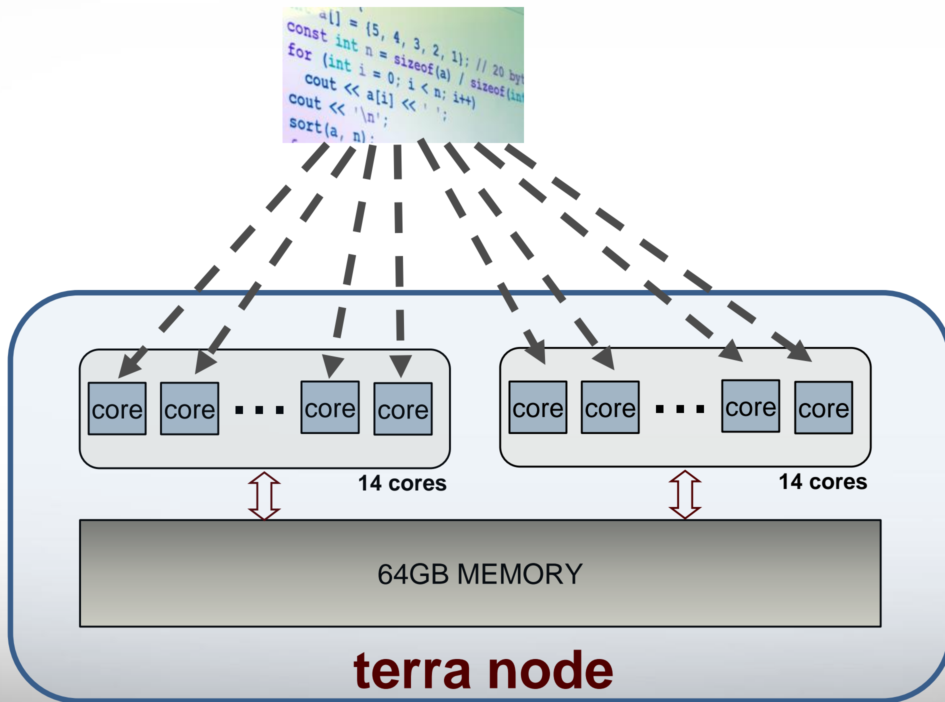


Basic Computer Architecture



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)

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)

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

C/C++ pragma format:

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

New line required

fortran directive format:

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

Not case sensitive

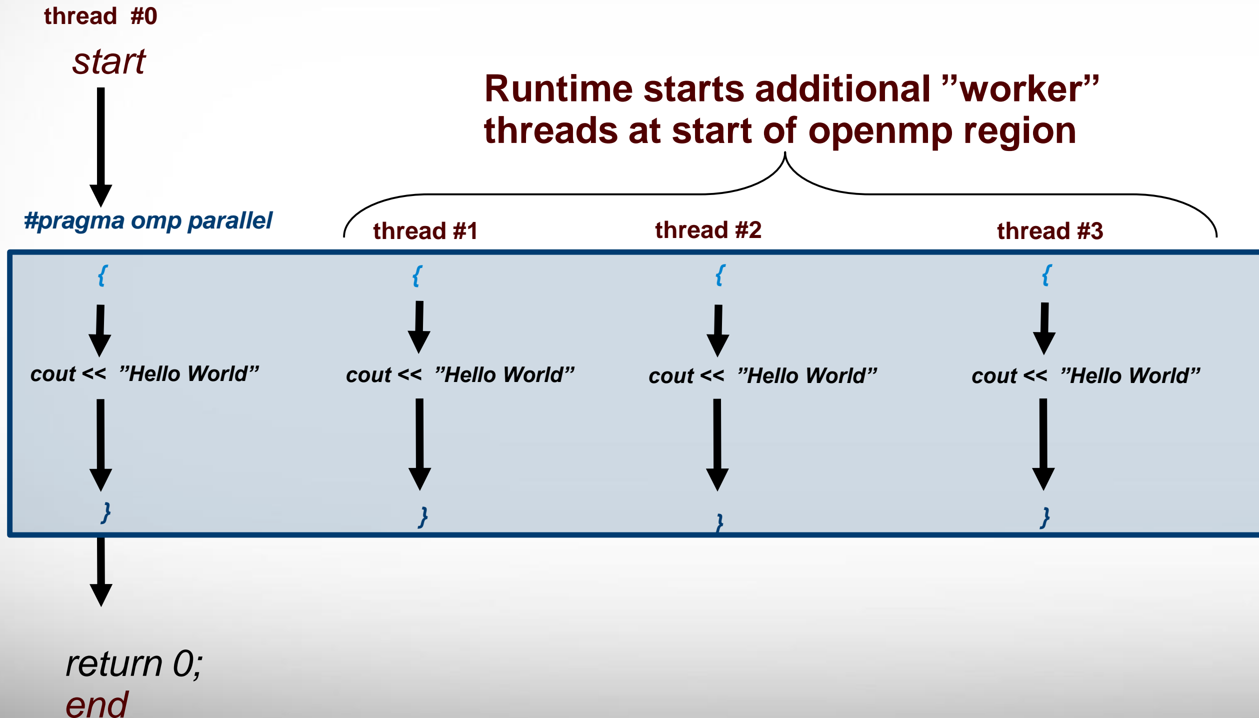
Starting Parallel Region

```
#pragma omp parallel
{
    // code block, will be
    // executed in parallel
}
```

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

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

Fork/Join




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

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

HelloWorld

SOURCE

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

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

pragma

directive

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

COMPILING

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
```

environmental variable

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.

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);
#pragma omp parallel
   :
```

???

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

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

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>
}
```

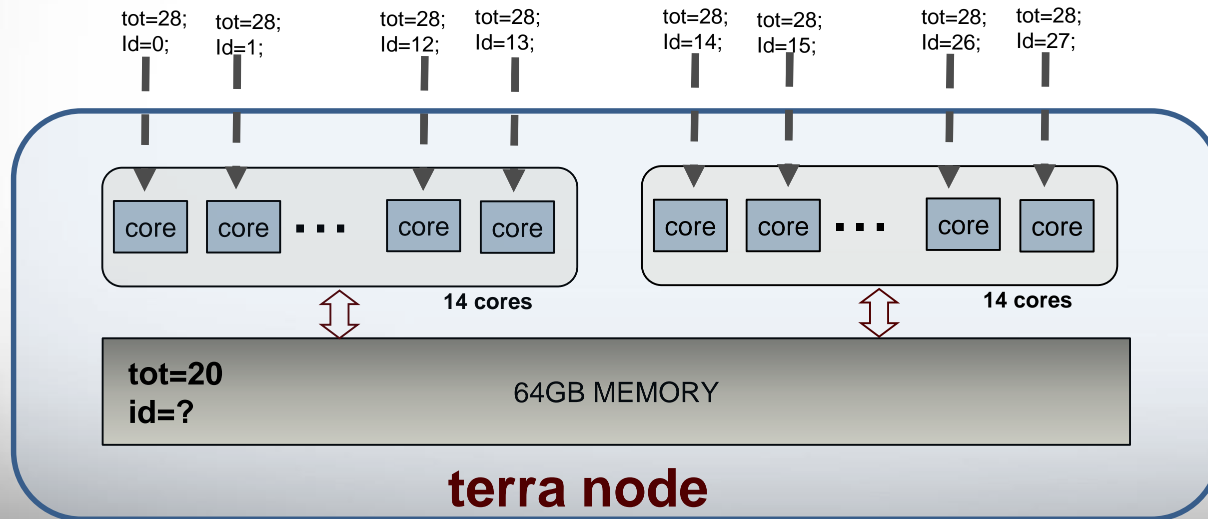
Exercise

Create an OpenMP program that does the following:

1) start parallel region
2) every thread prints it's id and total number of threads
3) close the parallel region

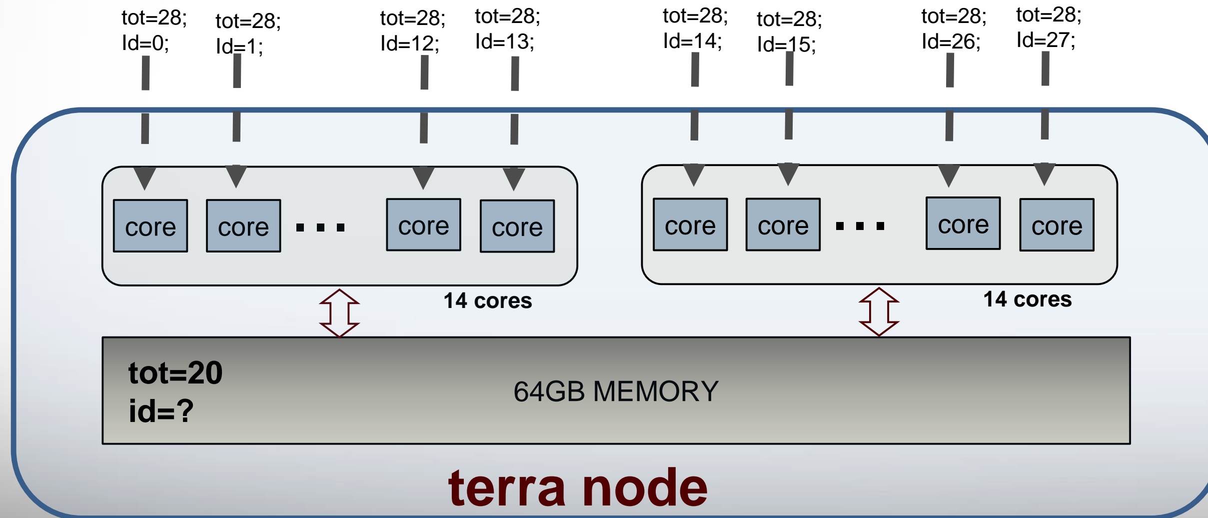
```
#pragma omp parallel
{
  tot = omp_get_num_threads();
  id = omp_get_thread_num();
}
```

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 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 (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

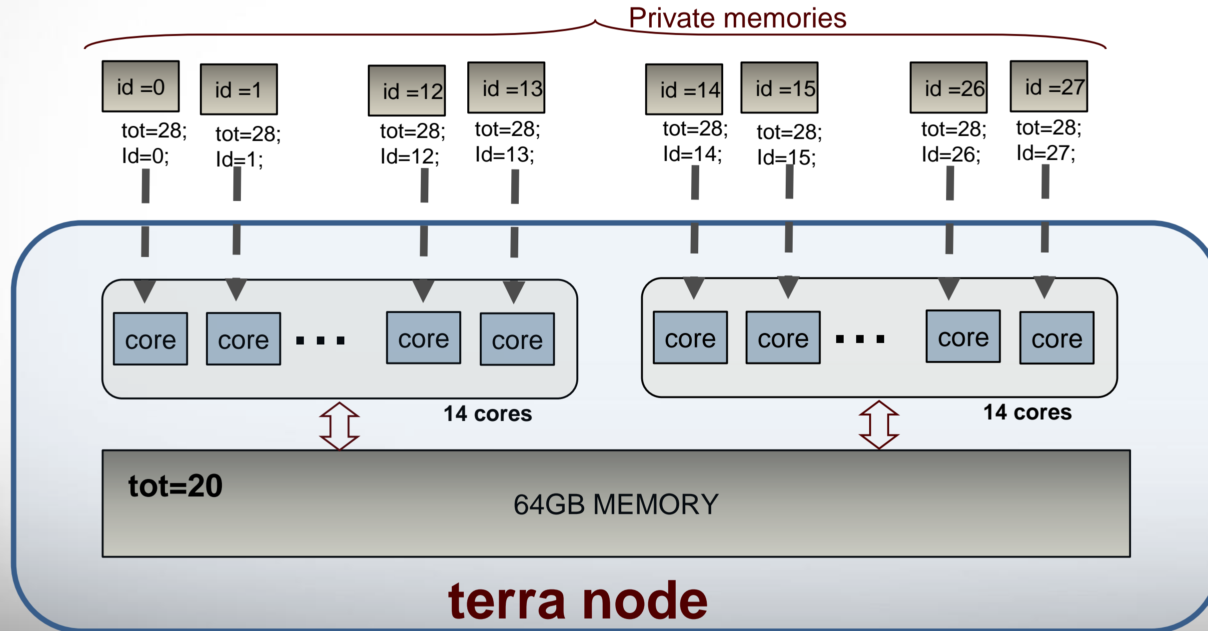
By default most variables in work sharing constructs are considered shared in OpenMP. Exceptions include index variables (Fortran, C/C++) and variables declared inside parallel region (C/C++).

Exercise

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

```
#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).

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(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***"
- Originally iteration variable could only be signed/unsigned integer variable.

Work Sharing Directives

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
}
```

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

Exercise

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

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 guranteed 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:        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

In a reduction, local copies of variable will be reduced into a global shared variable!

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

```
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*

Exercise

Create a subroutine that computes the dot product of two vectors. Use OpenMP pragmas (directives) to make it run in parallel.

Bonus Topic

(time permitted)

OpenMP Synchronization constructs

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 excute the code inside the block
- Execution of the block is serialized, only one thread at a time can execute the block
- Threads will wait at end of critical block until all threads have executed 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

After executing block, thread will wait until all other treads have finished.

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

Exercise

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.

Synchronization pragma

#pragma omp master (!$OMP MASTER)

- **ONLY** master threads will excute 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", "#pragma omp critical"

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

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:

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

This will only run the parallel region when n> 100000

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) {
        :
```

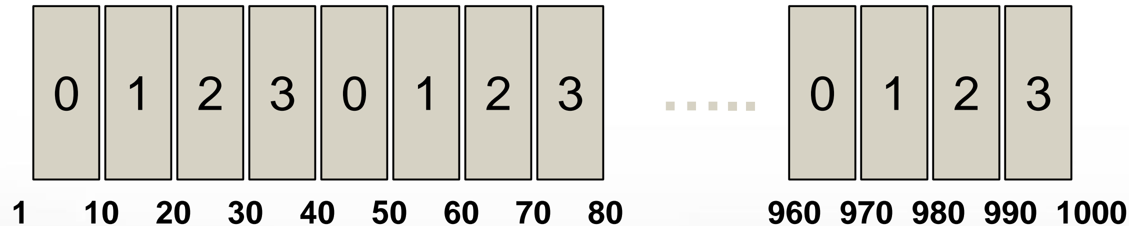
Although the OpenMP standard does not specify how a loop should be partitioned most compilers split the loop in N/p (N #iterations, p #threads) chunks by default.

Scheduling Clauses

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

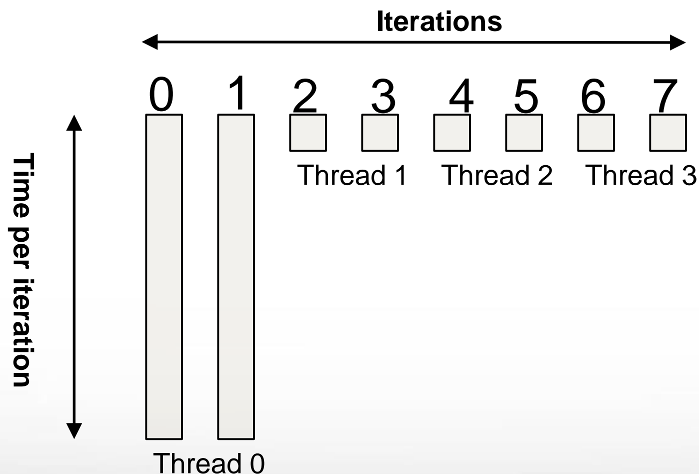
```
!$OMP PARALLEL DO SCHEDULE (STATIC,10)
DO i=1,1000
   :
ENDDO
!$OMP END PARALLEL DO
```

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



Scheduling Clauses

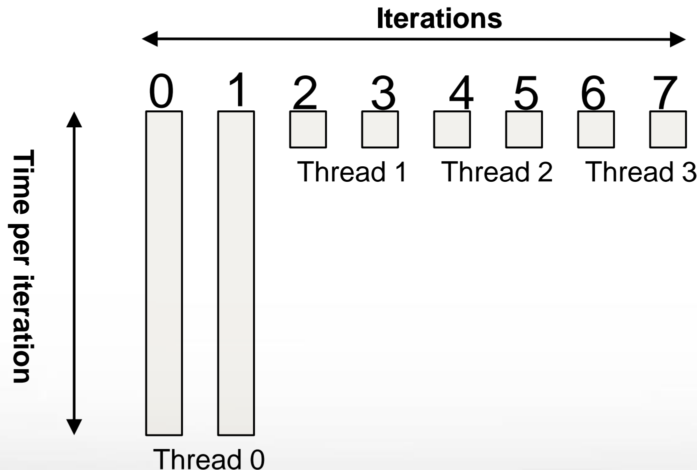
With static scheduling the number of iterations is evenly distributed among all openmp threads. This is not always the best way to partition. Why?



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

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

<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?

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

Work Sharing Directives

#pragma omp single (!$OMP SINGLE)

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

#pragma omp sections (!$OMP SECTIONS)

- Will execute all ”sections” concurrently

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

NOWAIT Clause

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