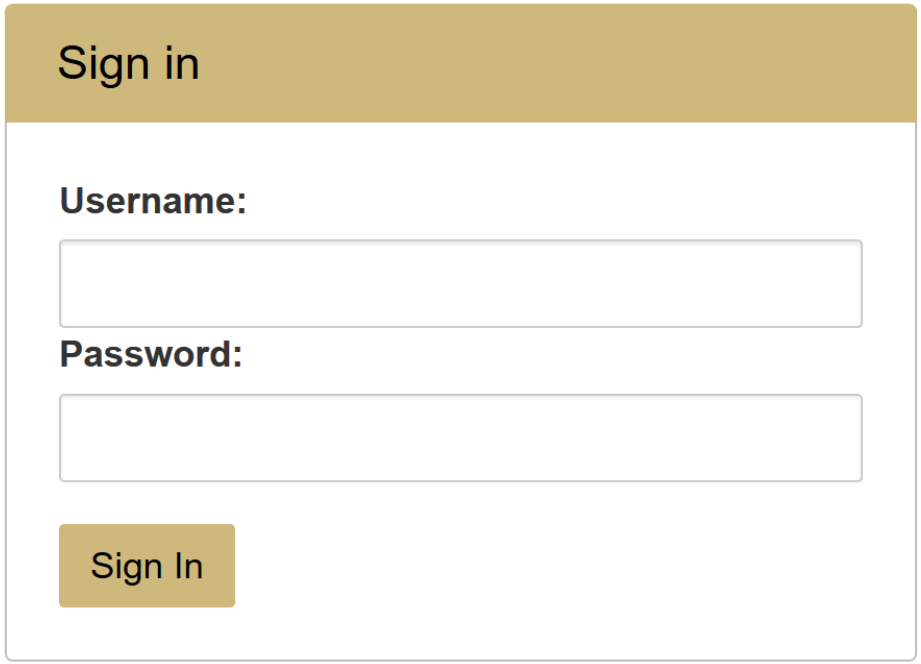
Nuts and Bolts of Parallel Programming

Nick Featherstone
feathern@colorado.edu
www.rc.colorado.edu

Slides:
https://github.com/ResearchComputing/Basics_Supercomputing

Before we Begin:
Log on to Sandstone

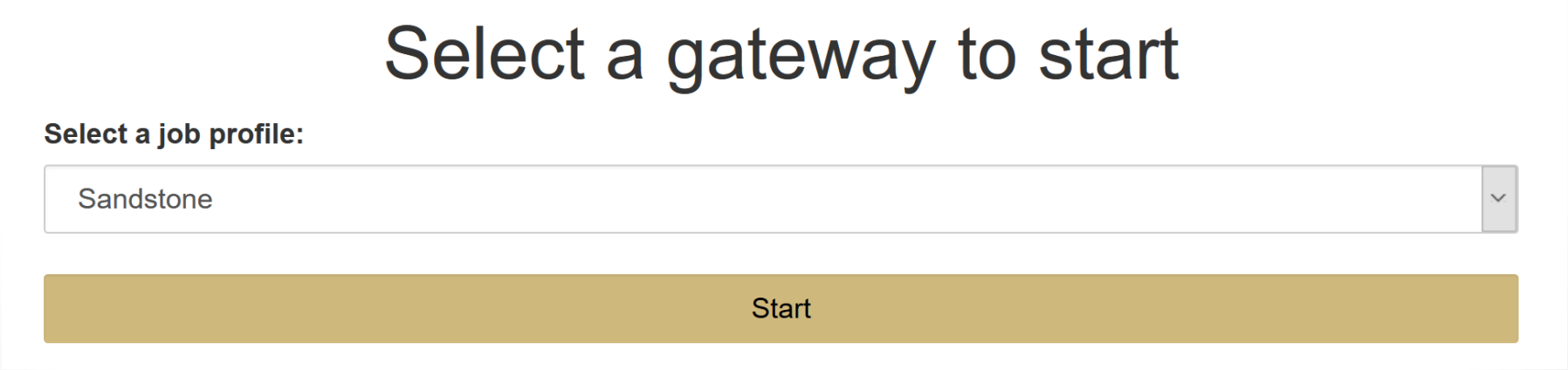
https://sandstone.rc.colorado.edu



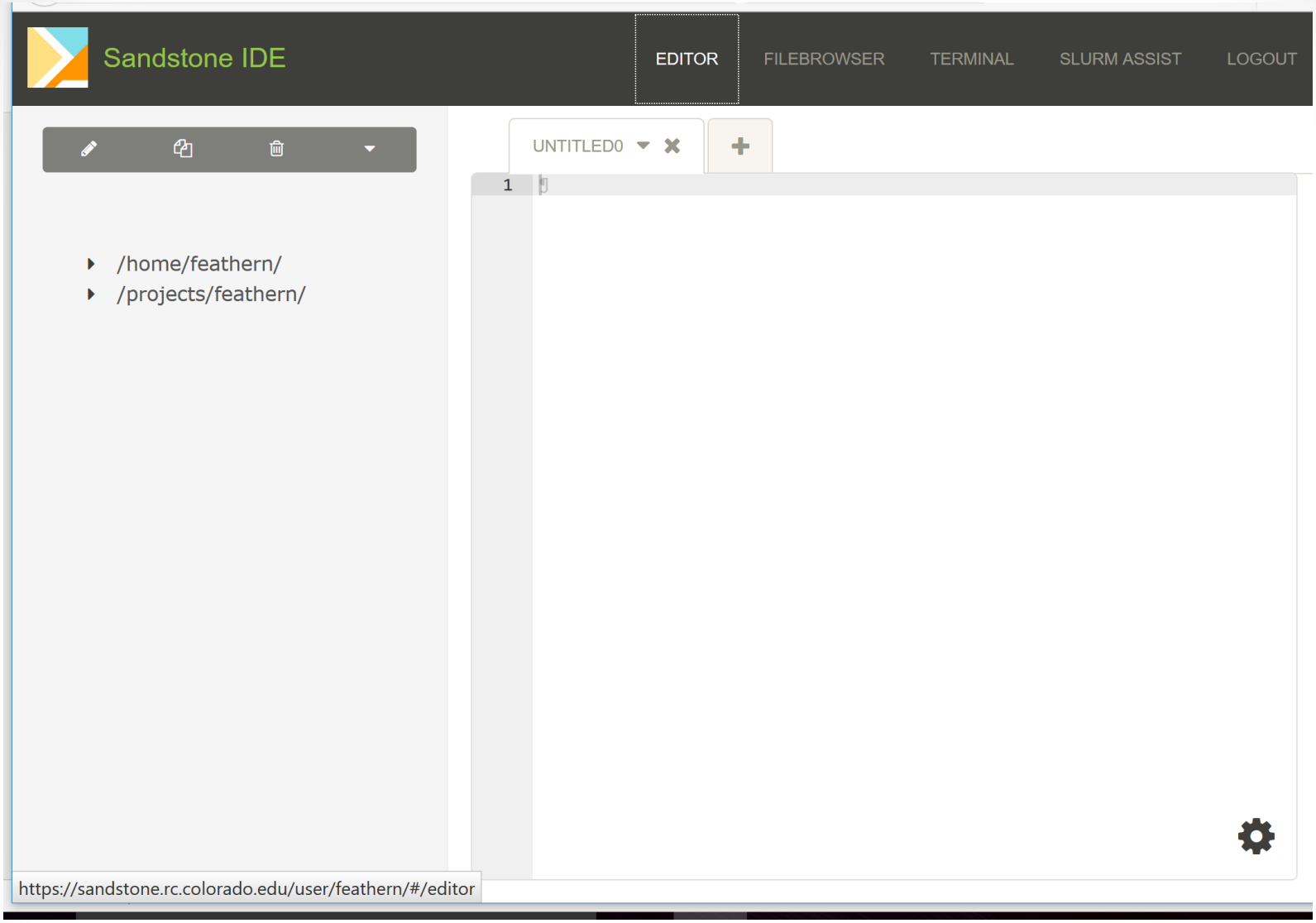
Use your tutorial credentials

Starting up Sandstone

Click the big green button



Choose "Sandstone" and click "Start"



Let's have a look

Initiating an Interactive Session

- As opposed to submitting job scripts, we will work interactively today. This approach is great for debugging.
- You will retain access to a single node of Summit throughout this tutorial.
- From the terminal window in your browser:

  module load slurm/summit

  sinteractive –N1 –n24 –t90
- Typing “exit” will leave the interactive session.

Parallel Programming Approaches

- Message Passing Interface (MPI)
  - Necessary for running on large supercomputers
  - Allows communication between different nodes via Ethernet/Infiniband/Omnipath etc. (distributed memory)
  - Versatile, but tedious to program

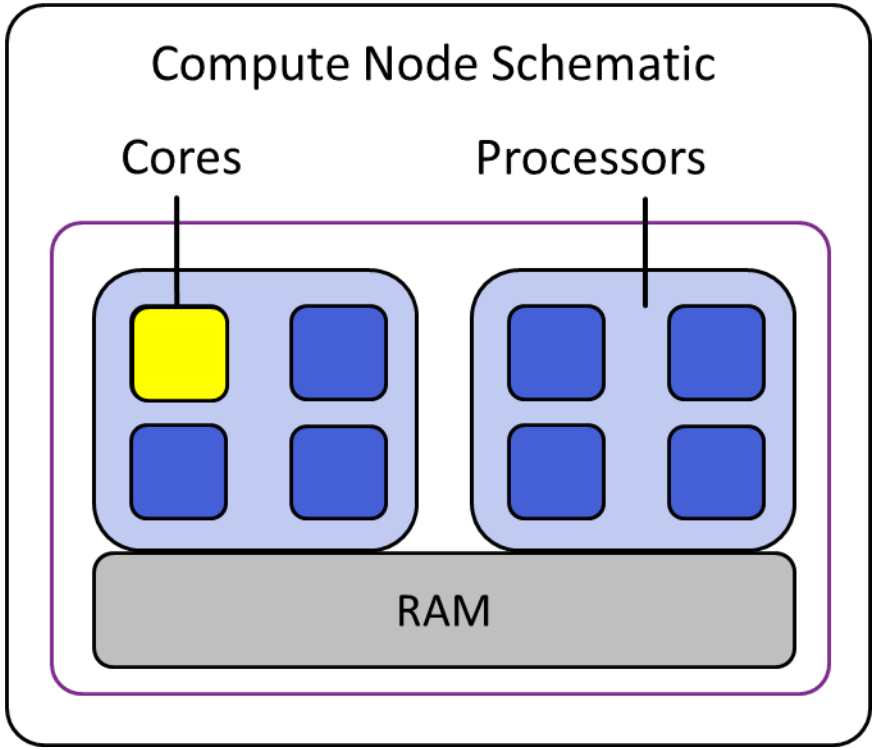
- OpenMP (Open Multi-processing)
  - Enables parallelism within a single node only (shared memory)
  - Relatively easy to program (directive-based)
  - Can be used in tandem with MPI

    (MPI between nodes; OpenMP within node)
  - Focus of today's session (many similar concepts to MPI)
  - No native Python support

WHY OPENMP: Serial Code Execution

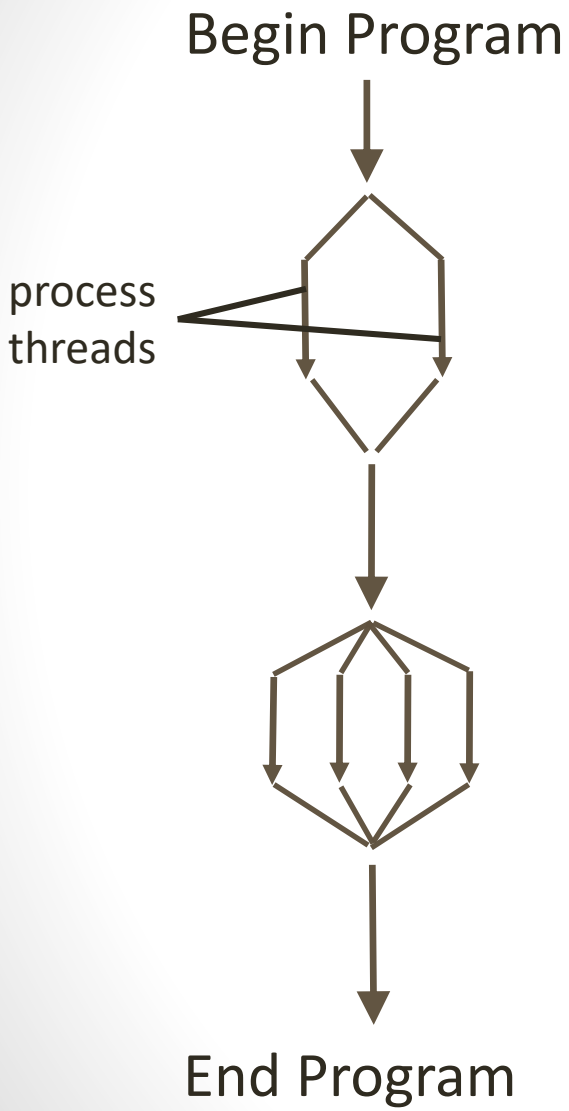
Begin Program

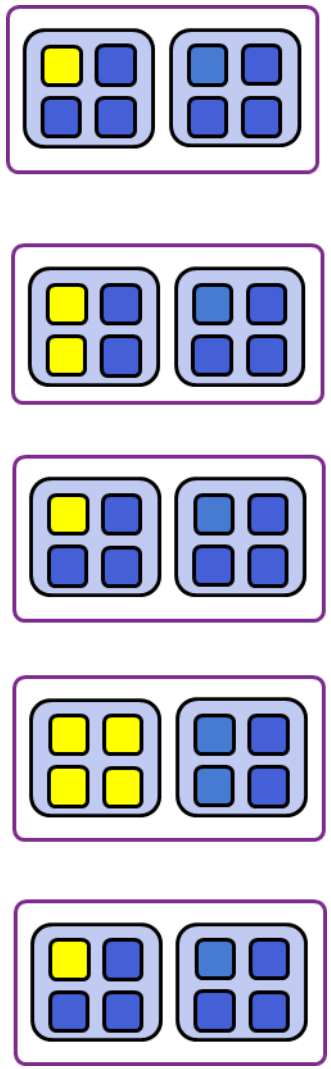
End Program



- Serial code runs on a single core
- Summit nodes have 24 cores…
- Wasted resources…

WHY OPENMP: Threaded Code Execution





- Fork/Join Model
- Portions in serial
- Others in parallel

Threads: copies of same code, running on multiple cores within same node.

WHY USE OPENMP?

- Rapid parallelization of serial code: 2x or more speedup
- Most desktops/laptops have multiple cores – you don't need a supercomputer.
- Memory considerations
- Message-count optimization

HOW DOES IT WORK?

- Add *directives* into existing serial code (fast to implement)

```
x = 0
Do i = 1 , N
        x = x + i**2
EndDo
```

Serial code

```
!$OMP PARALLEL DO
Do i = 1 , N
        x = x + i**2
EndDo
!$OMP END PARALLEL DO
```

Parallel code

Choose your path:

- Fortran and C/C++ examples are available
- Change to appropriate directory:
  - *C/C++*

    Day_Two/Nuts_and_Bolts/C++/OpenMP
  - *FORTRAN*

    Day_Two/Nuts_and_Bolts/Fortran/OpenMP
- A Python option is available for the MPI portion

OpenMP “Hello World”

Edit this file: Day_Two/
Nuts_and_Bolts / {Fortran, C++}/
omp_hello.{f90,cpp}

In the terminal window, compile and run the code… (next slide)

Compiling & Running with OpenMP

- Compile using the –qopenmp flag:

BOTH Fortran & C++

```
$ module load intel
```

C++

```
$ icpc  –qopenmp  omp_hello.cpp  –o   hello
```

FORTRAN

```
$ ifort  –qopenmp  omp_hello.f90  –o   hello
```

BOTH Fortran & C++

```
$ export OMP_NUM_THREADS=8
```

```
$ ./hello
```

Note:

OMP_NUM_THREADS: environment variable that controls the number of threads spawned by OpenMP within parallel regions.

If this variable is not set, OpenMP uses the maximum number of cores available (24 on a Summit node)

OpenMP Development Cycle for Today

1) Edit

2) Compile

```
$ ifort  –qopenmp  omp_hello.f90  –o  hello
```

3) Specify number of threads (cores) to use

```
$ export OMP_NUM_THREADS=8
```

*Only repeat this step when you wish to change the number of threads (compilation independent)

4) Run code

```
$ ./hello
```

5) Repeat

OMP_HELLO (Output)

```
[user0038@shas0701 OpenMP]$ ./hello2
 Hello world from the only processor here so far!
 Hello world from thread 6.
 Hello world from thread 2.
 Hello world from thread 4.
 Hello world from thread 1.
 Hello world from thread 3.
 Hello world from thread 5.
 8 threads are now active.
 Hello world from thread 0.
 Hello world from thread 7.
```

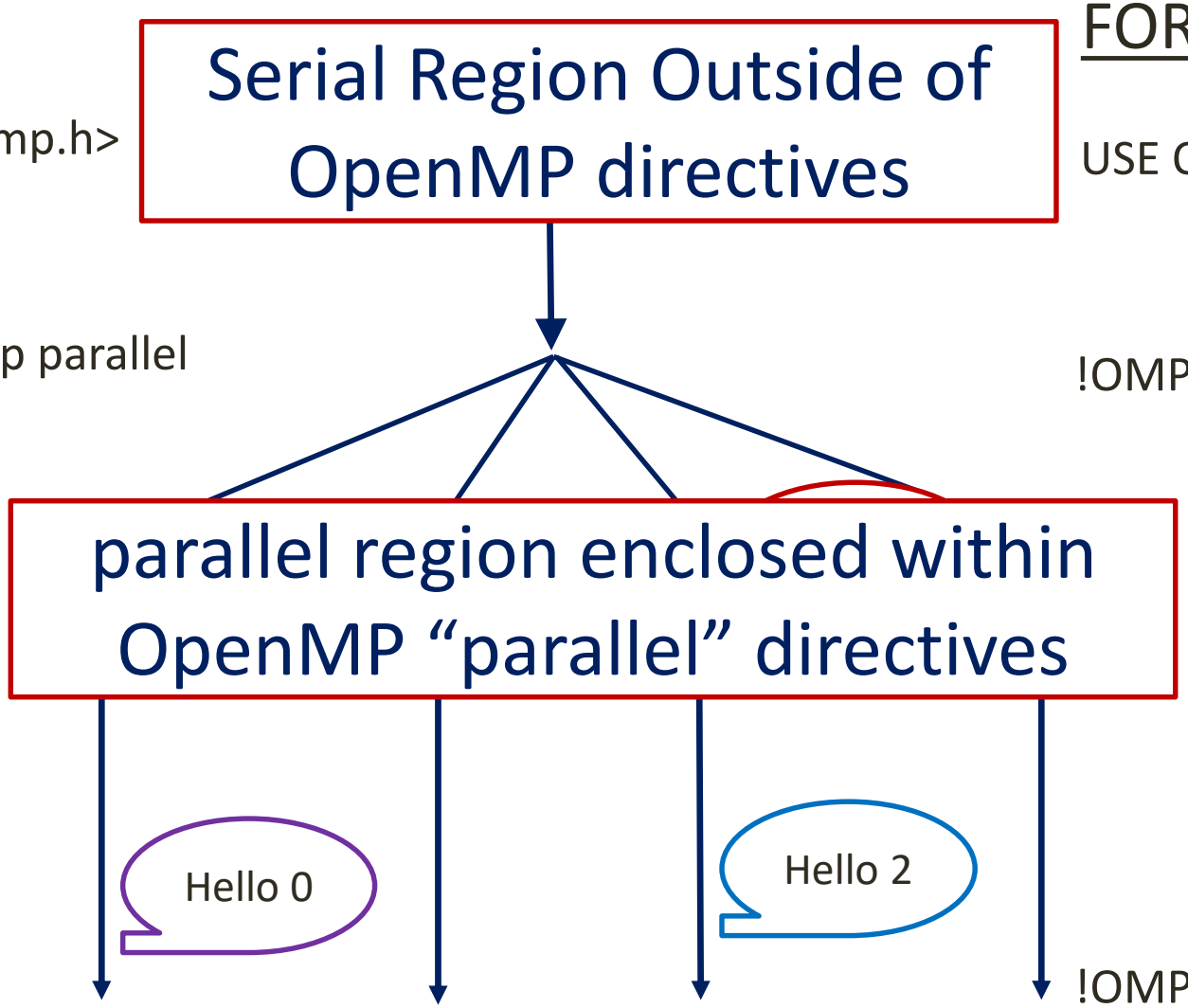
omp_hello schematic

C++

FORTRAN

#include <omp.h>

USE OMPLIB



#pragma omp parallel
{

!OMP PARALLEL

}

!OMP END PARALLEL

OpenMP in C++

Open ▾ | omp_hello1.cpp ~/parallel_programming/C++/OpenMP | Save

File Edit View Search Tools Documents Help

```
#include <stdio.h>
#include <omp.h>
//omp.h must always be included in c++ programs.

int main(int argc, char** argv) {

    // Print a hello world message from the main thread.
    printf("  Hello from the only process here so far!\n");

    // At this point in the program, only a single thread is active,
    // and the program behaves as a serial code.

    #pragma omp parallel
    {
    // Code inside this region runs in parallel.
    int thread_id; // Each thread has a unique copy of this variable

    //Each thread has a unique identifying number, which we store in thread_id
    thread_id = omp_get_thread_num();
    printf("  Hello from thread %d.\n",thread_id);

    }
}
```

include omp.h

OpenMP in C++

Open ▾ omp_hello1.cpp ~/parallel_programming/C++/OpenMP Save

File Edit View Search Tools Documents Help

```
#include <stdio.h>
#include <omp.h>
//omp.h must always be included in c++ programs.

int main(int argc, char** argv) {

    // Print a hello world message from the main thread.
    printf("  Hello from the only process here so far!\n");

    // At this point in the program, only a single thread is active,
    // and the program behaves as a serial code.

    #pragma omp parallel
    {
    // Code inside this region runs in parallel.
    int thread_id; // Each thread has a unique copy of this variable

    //Each thread has a unique identifying number, which we store in thread_id
    thread_id = omp_get_thread_num();
    printf("  Hello from thread %d.\n",thread_id);

    }
}
```

OMP Directive & brackets

OpenMP in C++

omp_hello1.cpp
~/parallel_programming/C++/OpenMP

```
#include <stdio.h>
#include <omp.h>
//omp.h must always be included in c++ programs.

int main(int argc, char** argv) {

    // Print a hello world message from the main thread.
    printf("  Hello from the only process here so far!\n");

    // At this point in the program, only a single thread is active,
    // and the program behaves as a serial code.

    #pragma omp parallel
    {
    // Code inside this region runs in parallel.
    int thread_id; // Each thread has a unique copy of this variable

    //Each thread has a unique identifying number, which we store in thread_id
    thread_id = omp_get_thread_num();
    printf("  Hello from thread %d.\n",thread_id);

    }
}
```

serial region

OpenMP in C++

Open ▾ | omp_hello1.cpp ~/parallel_programming/C++/OpenMP | Save

File Edit View Search Tools Documents Help

```
#include <stdio.h>
#include <omp.h>
//omp.h must always be included in c++ programs.

int main(int argc, char** argv) {

    // Print a hello world message from the main thread.
    printf("  Hello from the only process here so far!\n");

    // At this point in the program, only a single thread is active,
    // and the program behaves as a serial code.

    #pragma omp parallel
    {
    // Code inside this region runs in parallel.
    int thread_id; // Each thread has a unique copy of this variable

    //Each thread has a unique identifying number, which we store in thread_id
    thread_id = omp_get_thread_num();
    printf("  Hello from thread %d.\n",thread_id);

    }
}
```

parallel region

OpenMP in FORTRAN

omp_hello1.f90
~/parallel_programming/FORTRAN/OpenMP

Open Save

File Edit View Search Tools Documents Help

omp_hello1.cpp omp_hello1.f90

```
PROGRAM MAIN
    ! In FORTRAN, we must use the OMP_LIB module in o
    ! to access OpenMP functions like omp_get_num_thr
    USE OMP_LIB
    Implicit None
    Integer :: thread_id
    Character*8 :: thread_string

    ! At this point in the program, only a single thread is active,
    ! and the program behaves as a serial code.
    Write(6,*)"  Hello from the only process here so far!"

    !$OMP PARALLEL private(thread_id, thread_string)
    ! Code within the "$OMP PARALLEL" region runs in parallel
    ! using the number of threads specified via OMP_NUM_THREADS

    !Each thread has a unique identifying number, which we store in thread_id
    thread_id = omp_get_thread_num()
    Write(thread_string,'(i8)')thread_id
    Write(6,*)"  Hello from thread "//trim(adjustl(thread_string))//"."

    !$OMP END PARALLEL

END PROGRAM MAIN
```

Use Statement

OpenMP in FORTRAN

Open ▾ | omp_hello1.f90 ~/parallel_programming/FORTRAN/OpenMP | Save

File Edit View Search Tools Documents Help

omp_hello1.cpp | omp_hello1.f90

```fortran
PROGRAM MAIN
    ! In FORTRAN, we must use the OMP_LIB module in order
    ! to access OpenMP functions like omp_get_num_threads.
    USE OMP_LIB
    Implicit None
    Integer :: thread_id
    Character*8 :: thread_string

    ! At this point in the program, only a single thread is active,
    ! and the program behaves as a serial code.
    Write(6,*)"  Hello from the only process here so far!"

    !$OMP PARALLEL private(thread_id, thread_string)
    ! Code within the "$OMP PARALLEL" region runs in parallel
    ! using the number of threads specified via OMP_NUM_THREADS

    !Each thread has a unique identifying number, which we store in thread_id
    thread_id = omp_get_thread_num()
    Write(thread_string,'(i8)')thread_id
    Write(6,*)"  Hello from thread "//trim(adjustl(thread_string))//"."

    !$OMP END PARALLEL

END PROGRAM MAIN
```

OMP Directives

OpenMP in FORTRAN

omp_hello1.f90
~/parallel_programming/FORTRAN/OpenMP

Open Save

File Edit View Search Tools Documents Help

omp_hello1.cpp omp_hello1.f90

```fortran
PROGRAM MAIN
    ! In FORTRAN, we must use the OMP_LIB module in order
    ! to access OpenMP functions like omp_get_num_threads.
    USE OMP_LIB
    Implicit None
    Integer :: thread_id
    Character*8 :: thread_string

    ! At this point in the program, only a single thread is active,
    ! and the program behaves as a serial code.
    Write(6,*)"  Hello from the only process here so far!"

    !$OMP PARALLEL private(thread_id, thread_string)
    ! Code within the "$OMP PARALLEL" region runs in parallel
    ! using the number of threads specified via OMP_NUM_THREADS

    !Each thread has a unique identifying number, which we store in thread_id
    thread_id = omp_get_thread_num()
    Write(thread_string,'(i8)')thread_id
    Write(6,*)"  Hello from thread "//trim(adjustl(thread_string))//"."

    !$OMP END PARALLEL

END PROGRAM MAIN
```

Serial Region

OpenMP in FORTRAN

omp_hello1.f90
~/parallel_programming/FORTRAN/OpenMP

Open Save

File Edit View Search Tools Documents Help

omp_hello1.cpp | omp_hello1.f90

```fortran
PROGRAM MAIN
    ! In FORTRAN, we must use the OMP_LIB module in order
    ! to access OpenMP functions like omp_get_num_threads.
    USE OMP_LIB
    Implicit None
    Integer :: thread_id
    Character*8 :: thread_string

    ! At this point in the program, only a single thread is active,
    ! and the program behaves as a serial code.
    Write(6,*)"  Hello from the only process here so far!"

    !$OMP PARALLEL private(thread_id, thread_string)
    ! Code within the "$OMP PARALLEL" region runs in parallel
    ! using the number of threads specified via OMP_NUM_THREADS

    !Each thread has a unique identifying num                    hread_id
    thread_id = omp_get_thread_num()
    Write(thread_string,'(i8)')thread_id
    Write(6,*)"  Hello from thread "//trim(adjustl(thread_string))//"."

    !$OMP END PARALLEL

END PROGRAM MAIN
```

Parallel Region

Shared vs. Private

```
Write(6,*)"  Hello from the only process here so far!"

!Remember that everything out here is run in serial mode...

!$OMP PARALLEL private( thread_string, num_threads, thread_id)

! Note the use of the "private" clause above.  Each thread receives
```

"private" clause

- Variables are shared among threads unless declared as private.
- This can cause unexpected results if threads access same memory address.
- Variables can be explicitly listed as "shared" as well (stylistic)

```
!We might have regions of the code that we only execute if more than one thread is running.

END PROGRAM MAIN
```

Quick EXERCISE:

Remove thread_id from the private clause...
Recompile and run the code.
What happens?

Don't forget to change the program back!

Useful OpenMP functions:

- Use these within a parallel region

```
thread_id = OMP_GET_THREAD_NUM( )
```

- Retrieves unique numeric identifier for each thread

```
nthread = OMP_GET_NUM_THREADS( )
```

- Retrieves number of active threads

OMP_HELLO (Output)

```
[user0038@shas0701 OpenMP]$ ./hello2
 Hello world from the only processor here so far!
 Hello world from thread 6.
 Hello world from thread 2.
 Hello world from thread 4.
 Hello world from thread 1.
 Hello world from thread 3.
 Hello world from thread 5.
 8 threads are now active.
 Hello world from thread 0.
 Hello world from thread 7.
```

Seems out of order?

We can synchronize different threads using a ***barrier.***

OpenMP Barrier

Open this file: Day_Two/ Nuts_and_Bolts / {Fortran, C++}/ **omp_barrier.{f90,cpp}**

In the terminal window, compile and run the code…

OMP_BARRIER Output

```
user0038@tutorial-login:~/parallel_programming/C++/OpenMP
ile  Edit  View  Search  Terminal  Help
user0038@tutorial-login OpenMP]$ ./hello3
 Hello world from the only processor here so far!
 8 threads are now active.
 Hello world from thread 6.
 Hello world from thread 3.
 Hello world from thread 0.
 Hello world from thread 2.
 Hello world from thread 5.
 Hello world from thread 4.
 Hello world from thread 7.
 Hello world from thread 1.
user0038@tutorial-login OpenMP]$
```

Better!

What's changed?

omp_barrier.{cpp,f90}

```
if (num_threads > 1)
{
    if (thread_id == 0)
    {
        printf("  %d threads are now active.\n", num_threads);
    }

    // The new feature here is the use of "BARRIER," useful
    // in managing thread activity.  Execution of the parallel
    // threads pause at the barrier and resumes once all threads have
    // reached the barrier.
```

"barrier" directive

```
    #pragma omp barrier

    // Consider the loop below.  Where can we place another barrier to ensure
    // that
    for (in
    {
        if
        {

        }
    }
```

EXERCISE:

Place a barrier directive within the loop so that threads print their hello statement in ascending order based on thread ID.

Looping in OpenMP

- Most OpenMP efforts involve parallelizing loops.
- This is done via loop directives

```
!$OMP PARALLEL PRIVATE(i)

!$OMP DO
DO i = 1, 100
      x = x+i
ENDDO
!$OMP END DO

!$OMP END PARALLEL
```

Fortran

```
#pragma omp parallel private(i)
{
      #pragma omp for
      for (i=1,i<101;++i){
            x = x+i
      }
}
```

C++

Looping in OpenMP

- Parallel and for/do directives can be combined

```
!$OMP PARALLEL DO PRIVATE(i)
DO i = 1, 100
      x(i) = x(i)+i
ENDDO
!$OMP END PARALLEL DO
```

Fortran

```
#pragma omp parallel for private(i)
for (i=1,i<101;++i){
      x[i] = x[i]+i
}
```

C++

Looping in OpenMP

- Each thread handles a subset of the total iterations
- How iterations are distributed is controlled via schedule clause. Typically use *static* (default) or *dynamic*.

```
!$OMP PARALLEL DO PRIVATE(i) SCHEDULE(STATIC)  OR  SCHEDULE(DYNAMIC)
DO i = 1, 100
      x = x+i
ENDDO
!$OMP END PARALLEL DO
```

- Static scheduling:
  - Thread 0 gets iterations 1-10
  - Thread 1 gets 11-20, etc.

- Dynamic scheduling:
  - Thread 0 gets iterations 1,3,5,7
  - Thread 1 gets iterations 2,4,6,8 etc.

Looping Considerations

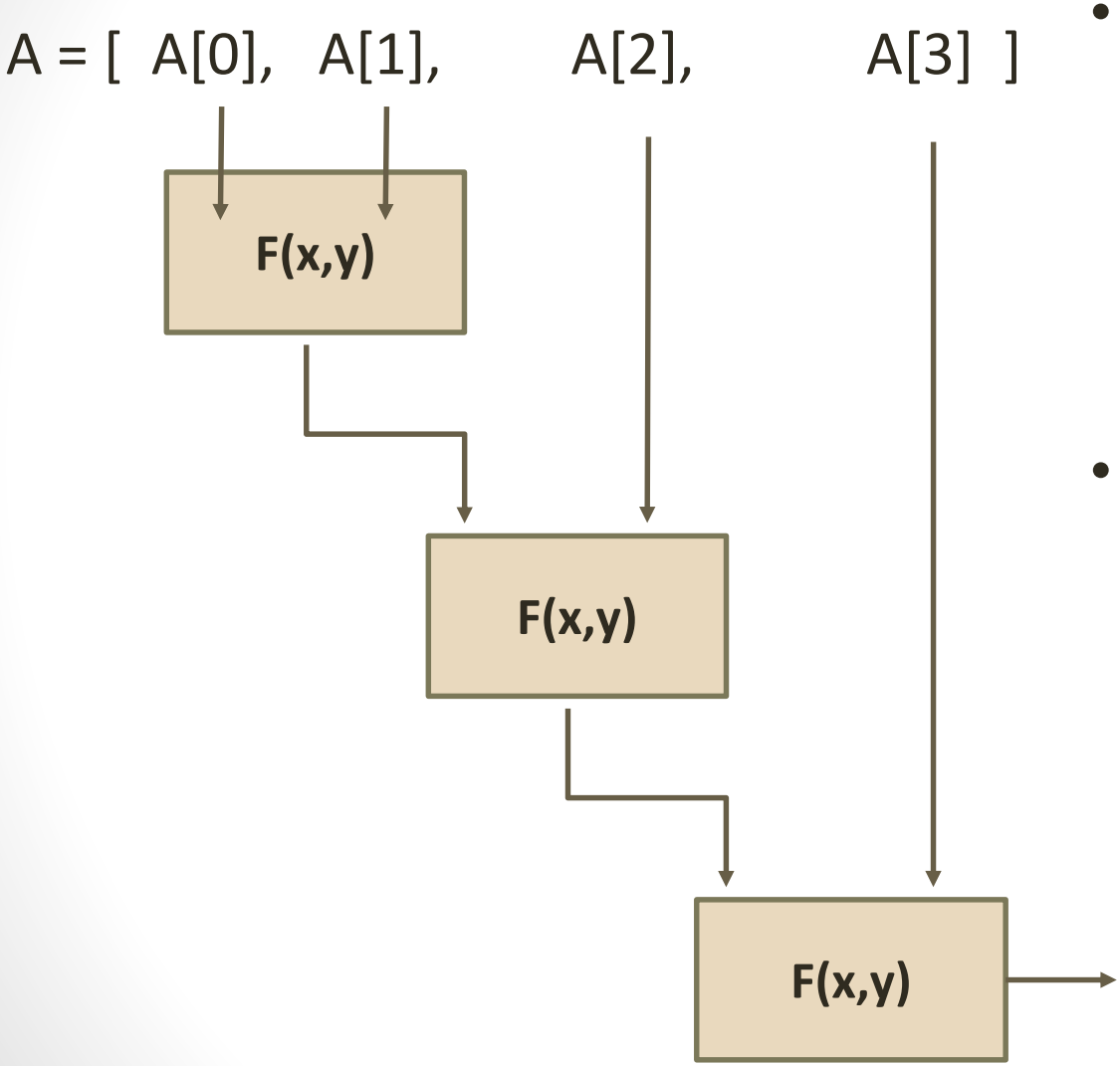
- Parallelize loops in with each iteration is independent

```
!$OMP PARALLEL DO PRIVATE(i)
DO i = 1, 100
      x(i) = x(i)+i
ENDDO
!$OMP END PARALLEL DO
```

```
!$OMP PARALLEL DO PRIVATE(i)
DO i = 1, 100
      x = x + i
ENDDO
!$OMP END PARALLEL DO
```

- The left side will parallelize well.
- The right side will not. Why?
- What if we want to parallelize a sum…

Common Parallel Operation: *Reduction*



- Sequential function evaluation using results from previous evaluations in tandem with new inputs
- Common forms:
  - Summation
  - Products
  - Min/Max

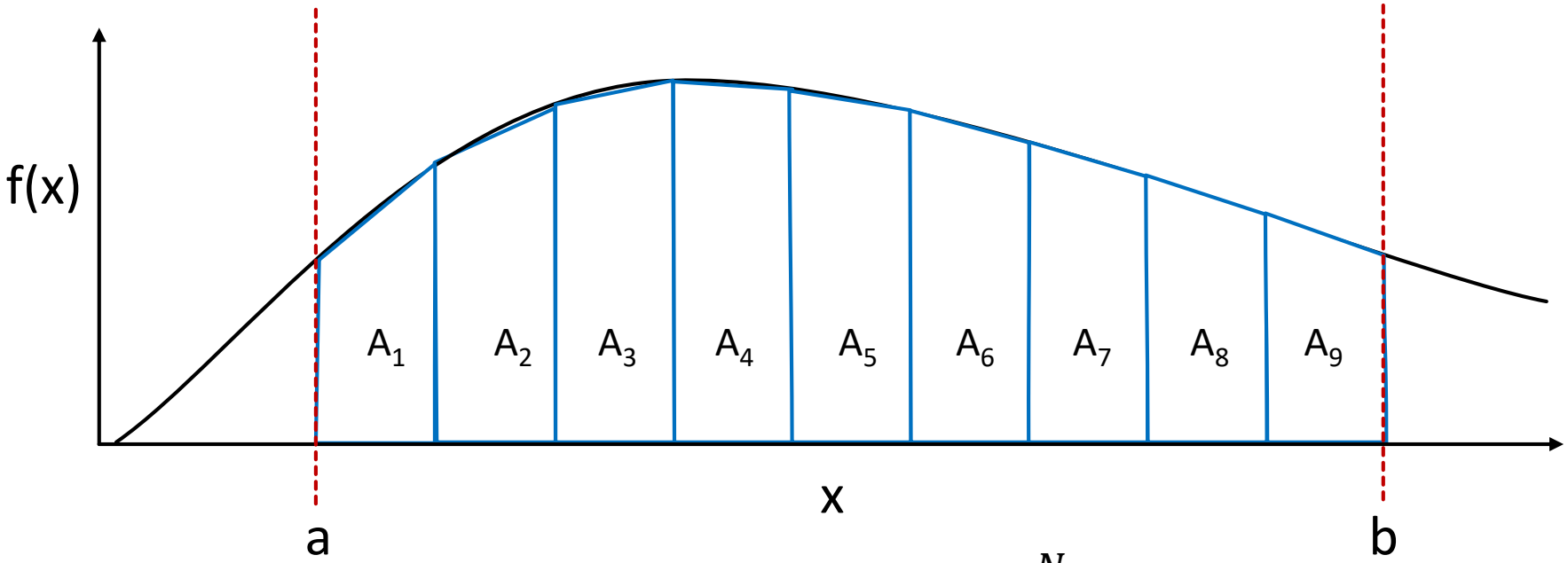
Reduction in OpenMP

- Loops with interdependent iterations can be parallelized by adding the reduction clause

```
!$OMP PARALLEL DO PRIVATE(i) REDUCTION(+:x)
DO i = 1, 100
      x = x + i
ENDDO
!$OMP END PARALLEL DO
```

- Value of x is added up across all threads at end of loop
- Some other reduction operations:
  - Multiplication: (*:x)
  - Minimum: (min:x)
  - Maximum: (max:x)

Reduction Application: Integration



$$Area = \int_a^b f(x)dx \approx \sum_{i=1}^{N} A_i$$

OpenMP Reduction

Edit this file: Day_Two/
Nuts_and_Bolts / {Fortran, C++}/
omp_reduction.{f90,c}

Compile and run the code.

```
export OMP_NUM_THREADS=4
ifort –qopenmp omp_reduction.f90 –o reduction
./reduction
```

Exercise: Reduction

Edit this file: Day_Two/
Nuts_and_Bolts / {Fortran, C++}/
omp_trapezoid_exercise.{f90,cpp}

Compile and run the code.

```
export OMP_NUM_THREADS=4
ifort –qopenmp omp_trapezoid_exercise.f90 –o trap
./trap
```

The program functions, but it isn't parallel yet…

Examine the loop in the trapezoid_int function

```
    //#pragma omp parallel for private() shared() reduction(+:)
    for (i=1; i<ntrap-1; ++i)
    {
        x = a+i*h;
        integral = integral+myfunc(x);
    }

    integral = integral*h;
    return integral;
}
```

Uncomment

What variables should we include in the private, shared, and reductions clauses?

Modify the code appropriately, compile, and run.

Reduction Solution

```
    #pragma omp parallel for private(i,x) shared(h,ntrap) reduction(+:integral)
    for (i=1; i<ntrap-1; ++i)
    {
        x = a+i*h;
        integral = integral+myfunc(x);
    }

    integral = integral*h;
    return integral;
}
```

- Variables that change and which should be private:
  - i
  - x
  - integral
- The *integral* variable should be reduced

TIMING

- The point of all this was to make our calculations faster. Did we succeed?
- Change ntests from 2 to 10,000
- Time code for different numbers of threads:

```
$ export OMP_NUM_THREADS=2  (and 4, 8, 12, 24)
$ time ./omp_trapezoid_solution
```

Your run took this long →

Resource usage →
(threads X wall time)

```
$ real  0m2.600s
$ user 0m5.200s
$ sys   0m0.005s
```

A Quick Look at MPI

I mentioned programming MPI could be tedious…

What information do we need to know to exchange messages across the network?

- Which process is sending the message?
- Where is the data on the sending process?
- What kind of data is being sent?
- How much data is there?
- Which process is going to receive the message?
- Where should the data be stored on the receiving process?
- What amount of data is the receiving process prepared to accept?

Message Passing Syntax Example

- call MPI_SEND(

message,	e.g., my_partial_sum,
count,	number of values in msg
data_type,	e.g, MPI_DOUBLE_PRECISION,
destination,	e.g., myid + 1
tag,	some info about msg, e.g., store it
communicator,	e.g., MPI_COMM_WORLD,
ierr	error tag (return value)

)

Quite a bit to specify…

MPI Examples:

- Initialization: mpi_hello.{f90,cpp}
- Barrier: mpi_barrier.{f90,cpp}
- Reduction: mpi_reduction.{f90,c}
- Message Passing: mpi_messages.{f90,cpp}

- Let’s have a look…

Compiling & Running MPI Programs

BOTH Fortran & C++

```
$ module load intel impi
```

C++

```
$ mpicc   mpi_hello.cpp  –o   hello
```

FORTRAN

```
$ mpif90 mpi_hello.f90  –o   hello
```

BOTH Fortran & C++

```
$ mpiexec –np 4 ./hello
(or –np 8, -np 16, etc.)
```

Questions?

- Email rc-help@colorado.edu

- Link to survey on this topic:

http://tinyurl.com/curc-survey16

Speaker: Nick Featherstone
Title: Nuts and Bolts of Parallel Programming
July 2017 BSW

- Slides:
https://github.com/ResearchComputing/Basics_Supercomputing