# HOW TO PARALLEL PROGRAM

## … IN 90 MINUTES OR LESS

NICK FEATHERSTONE

2017 BASICS OF SUPERCOMPUTING BOOTCAMP

UNIVERSITY OF COLORADO, BOULDER

Who are we?
Why are we here?

# Why am I here?

Advice on:

- Parallel programming/design
- Optimization
- Compiler options/porting

feathern@colorado.edu

# IN THIS SESSION:

- *Exposure* to parallel programming
- Introduction to **OpenMP** and **MPI** concepts
- *Example-based*: FORTRAN, C++, and Python
- Assumptions:
  - You know how to program (even just a little)
  - You have never parallel programmed

# OUTLINE

I. Preliminaries
II. OpenMP with C++ and FORTRAN
III. MPI with C++, FORTRAN, and Python

# I. Preliminaries

# THIS INTRODUCTION IS NON-EXHAUSTIVE!

- MPI reference: https://computing.llnl.gov/tutorials/mpi/
- OpenMP reference: https://computing.llnl.gov/tutorials/openMP/

# BEFORE WE BEGIN:

- We will be doing a lot of development work:
  - editing code
  - compiling/debugging code
  - running code
- Well-suited to *interactive* sessions

- Open TWO terminal windows
- Within each window, ssh into the login node:

```
$ ssh –l name tutorial-login.rc.colorado.edu
```

# TERMINAL WINDOW 1:

- This is where we will RUN our code:
  - We will use an interactive session (no jobscripts)
  - mpirun commands etc. executed manually
- Execute these two commands:

```
$ module load slurm/summit
$ sinteractive –N1 –n24  -t90 --exclusive -Atutorial1
```

- Once your prompt changes to shasXXXX, type:

```
$ module unload all
$ module load intel
$ module load impi
```

# INTERACTIVE SESSIONS:

- Great for development work (not production):
  - When code exits, session remains active
  - Compile-run-debug, recompile-rerun-debug cycle
- Accessed through sinteractive command:

```
$ sinteractive –N1 –n24  -t90 --exclusive -Atutorial1
```

- Options used:
  - -N1 : request 1 node
  - -n24 : request 24 tasks (cores) per node
  - -t90 : request a 90 minute session
  - --exclusive : do not share nodes with other users
  - -Atutorial1 : charge job to account named tutorial1
- Typing “exit” will end the session

# INTERACTIVE SESSION VS. BATCH SCRIPT

```
$ sinteractive –N1 –n24  -t90 --exclusive -Atutorial1
```

```
$ mpirun –np 5 ./hello1
```

Equivalent

**jobscript.sh**

```
#!/bin/bash
#SBATCH -N 1                          # Number of nodes
#SBATCH –n 24                         # 24 tasks per node
#SBATCH --time=1:30:00                # Max walltime
#SBATCH --job-name=hello              # Job submission name
#SBATCH --reservation=tutorial1       # Reservation name
#SBATCH --exclusive

mpirun –np 5 ./hello1
```

```
$ sbatch jobscript.sh
```

# … OR WITH OPENMP…

```
$ export OMP_NUM_THREADS=24
```

```
$ ./hello1
```

Equivalent

**jobscript.sh**

```
#!/bin/bash
#SBATCH -N 1                          # Number of nodes
#SBATCH –n 24                         # 24 tasks per node
#SBATCH --time=1:30:00                # Max walltime
#SBATCH --job-name=hello              # Job submission name
#SBATCH --reservation=tutorial1       # Reservation name
#SBATCH --exclusive
export OMP_NUM_THREADS 24
./hello1
```

# TERMINAL WINDOW 2:

- Here, we will EDIT and COMPILE our code:
  - EDITORS: **nano,** emacs, and vi
- From login, we ssh into a compile node:

```
$ ssh scompile
```

- When prompt changes to shasXXXX, type:

```
$ module unload all
$ module load intel
$ module load impi
```

# QUICK NANO SURVIVAL TIPS

- We will use nano as our editor of choice
- To open a file from shell prompt: nano filename
- Some useful commands from within nano:
  - ctrl + o - save changes
  - ctrl + x - exit
  - ctrl + k - cut
  - ctrl + u - paste

# UNTAR THE TUTORIAL FILES:

- From within your home directory, type:

```
$ tar –xvf parallel_programming.tar
```

# CHOOSE YOUR PATH:

- We begin with OpenMP
- Within EACH window, cd into appropriate directory:
  - C++ : parallel_programming/C++/OpenMP
  - FORTRAN : parallel_programming/FORTRAN/OpenMP

- A Python option is available for the MPI portion
- Change to your compile window now

# THE MAKEFILE:

- Codes in this session are compiled using a Makefile
- Compiler commands specified within this file

```
$ more Makefile
```

```
CC = icc
CFLAGS = -fopenmp -O2

hello1:
    $(CC) $(CFLAGS) omp_hello1.cpp -o hello1
hello2:
    $(CC) $(CFLAGS) omp_hello2.cpp -o hello2
hello3:
    $(CC) $(CFLAGS) omp_hello3.cpp -o hello3
trapezoid:
    $(CC) $(CFLAGS) omp_trapezoid.cpp -o trapezoid
all: hello1 hello2 hello3 trapezoid
clean:
    rm -f hello1 hello2 hello3 trapezoid
```

C++ OpenMP Makefile

# MAKEFILE USAGE (TRY THIS):

- Initially, our directory contains:

```
$ ls
```

  - the Makefile
  - .f90 or .cpp source files
  - No programs

- To build all programs within a directory:

```
$ make all
$ ls
```

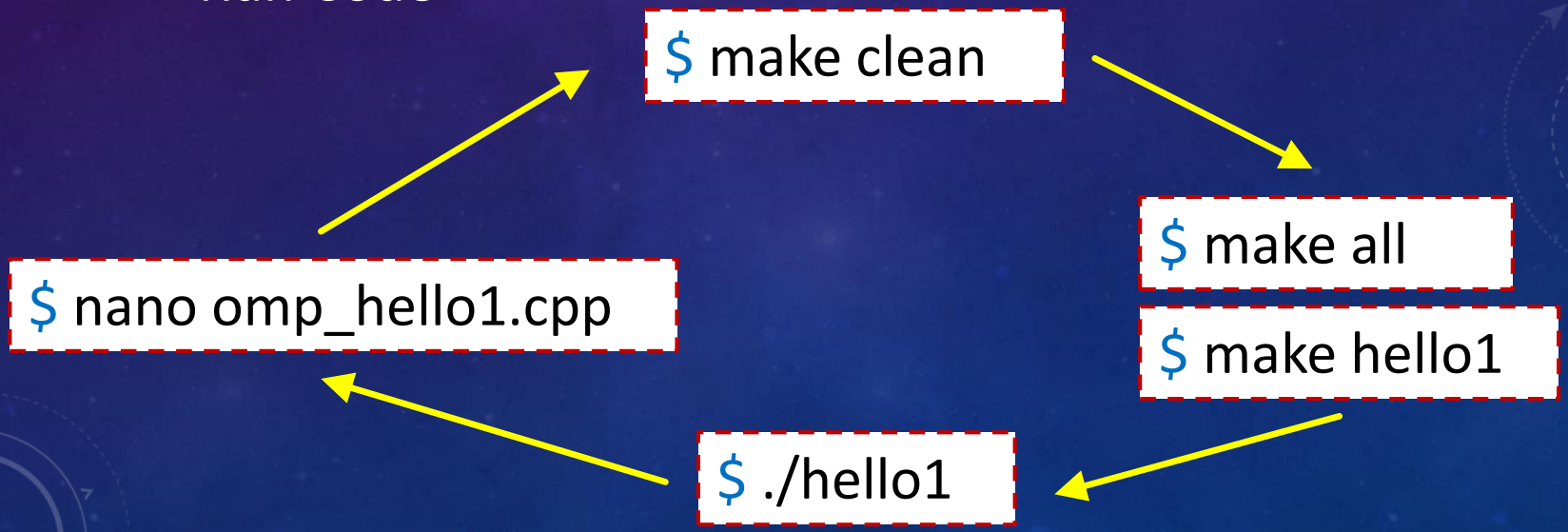
- To clean out compiled codes:

```
$ make clean
$ ls
```

Try this in your **COMPILE** window

# MAKEFILE USAGE

- Programs must be removed before recompiling:
- Development cycle:
  - Edit source code
  - Make clean
  - Make all  OR Make hello1, hello2 etc.
  - Run Code



# II. OpenMP

# WHAT IS OPENMP?

- An application programming interface (API) that supports multi-platform shared memory multiprocessing … Wikipedia

**OR**

- A means of using the full resources of a single node.

# WHY SHOULD YOU USE IT?

- Fast parallelization of serial code: 2x or more speedup
- Most desktops/laptops have multiple cores
- Memory considerations
- Message-count optimization

# COMPILING WITH OPENMP

- The compiler needs to be told to look for OpenMP instructions in a program.
- Accomplished through –fopenmp flag:

C++

```
$ icpc  –fopenmp  omp_hello1.cpp  –o  hello1
```

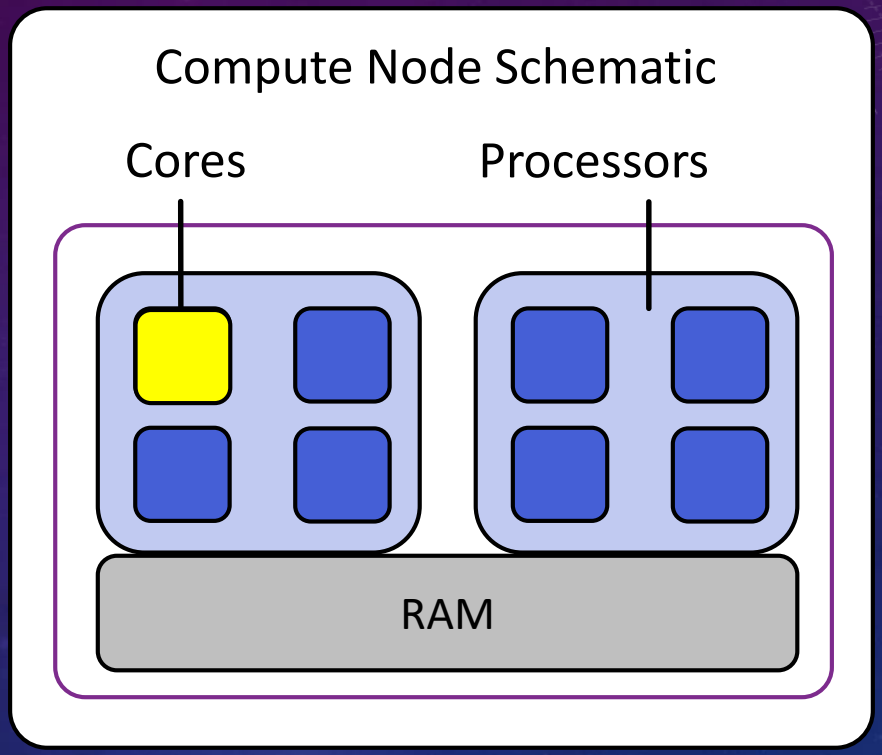
FORTRAN

```
$ ifort  –fopenmp  omp_hello1.f90  –o  hello1
```

The Makefiles are already set up to do this for you…

# WHY OPENMP: SERIAL CODE EXECUTION

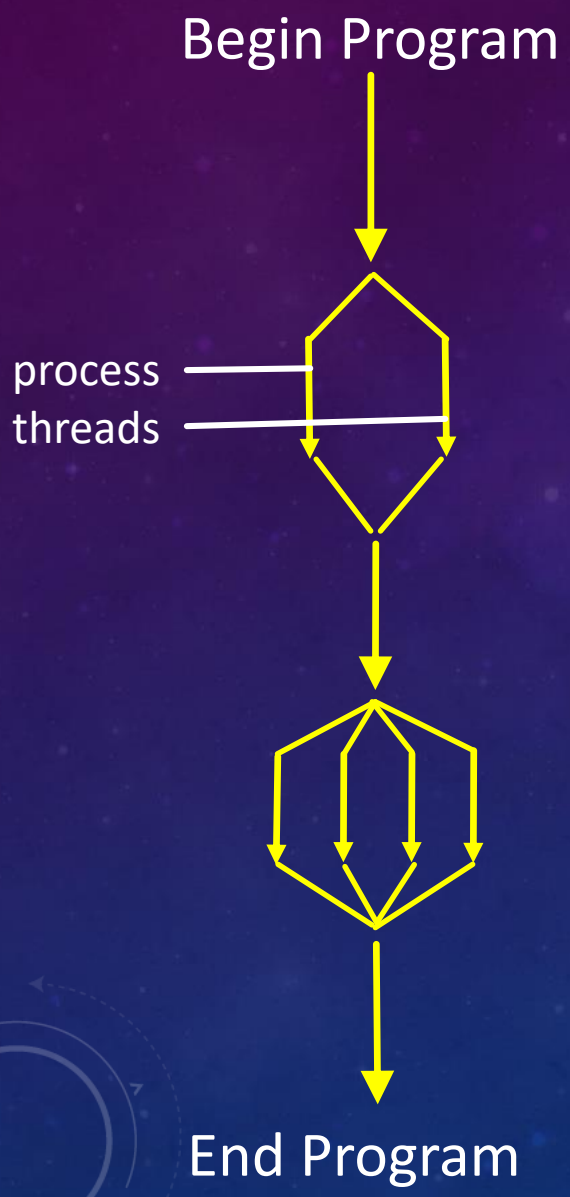
Begin Program

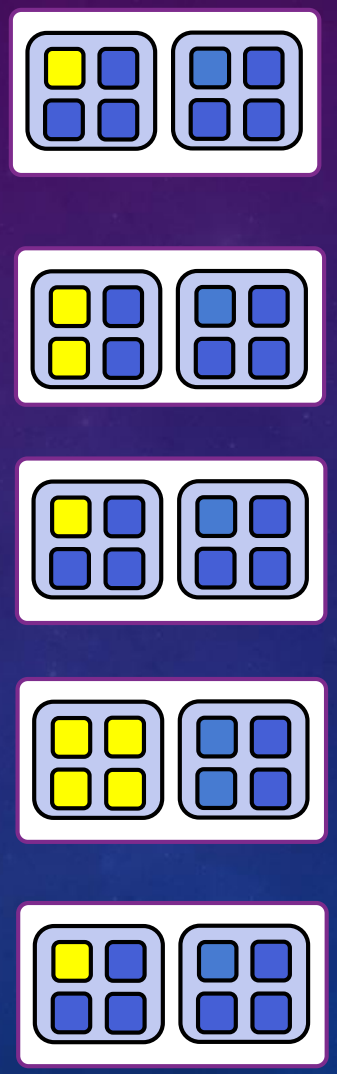



End Program

- 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.

# OPENMP: HELLO WORLD

**1. In the COMPILE window**

```
$ make clean
$ make all
```

**2. In the RUN window**

```
$ export OMP_NUM_THREADS=2
$ ./hello1
```

**3. In the RUN window**

```
$ export OMP_NUM_THREADS=5
$ ./hello1
```

**4. In the COMPILE window**

```
$ nano omp_hello1.cpp
$ nano omp_hello1.f90
```

**OMP_NUM_THREADS:** environment variable that controls the number of threads spawned by OpenMP within parallel regions. Defaults to number of cores on node (24 on Summit)

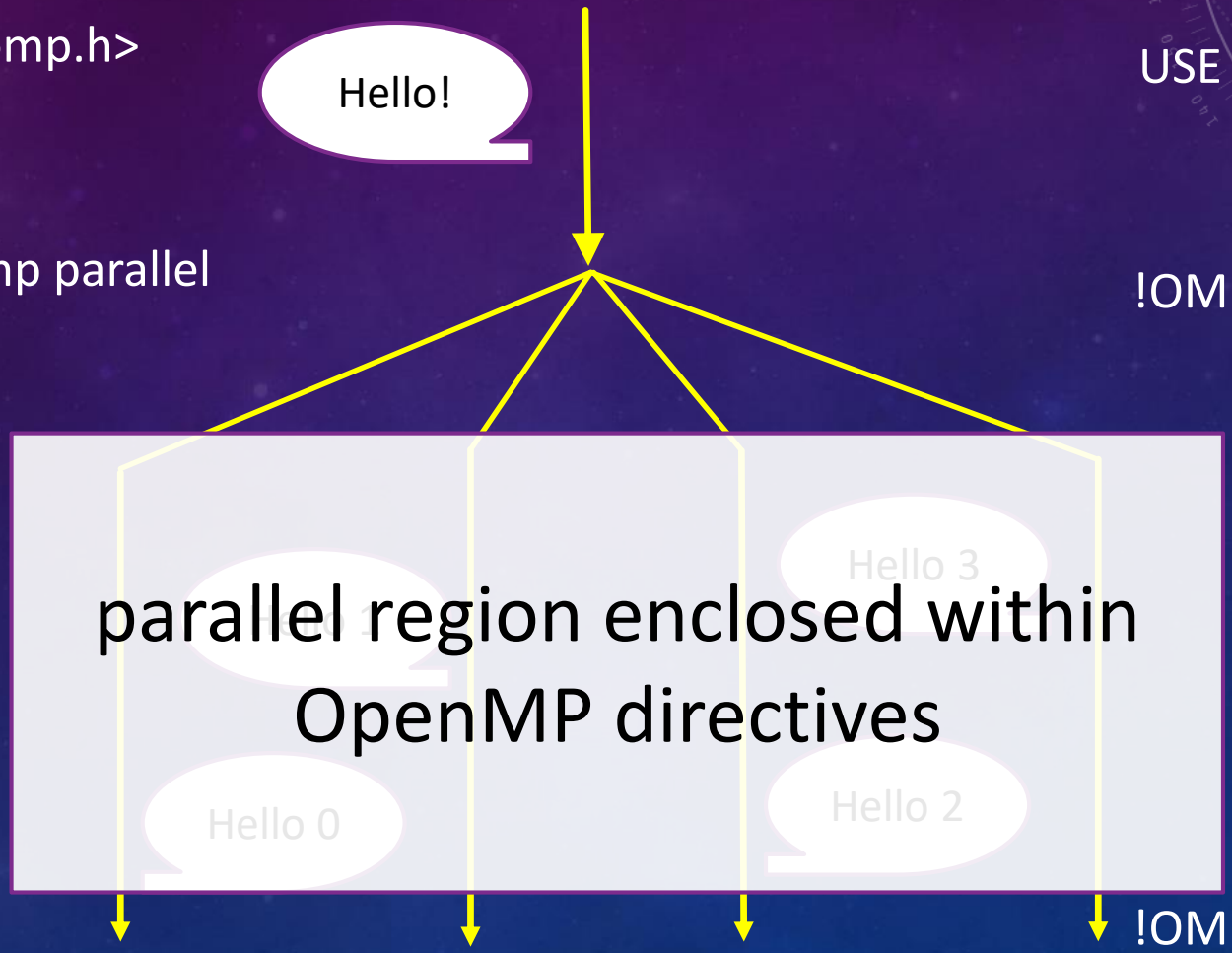
# HELLO1 SCHEMATIC

## C++

```
#include <omp.h>

#pragma omp parallel
{

}
```

## FORTRAN

```
USE OMPLIB

!OMP PARALLEL

!OMP END PARALLEL
```



# 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);

    }
}
```

# 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++

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);

    }
}
```

**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

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
```

# 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 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

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 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**

# USEFUL FUNCTION:

```
thread_id = OMP_GET_THREAD_NUM( )
```

- Use within a parallel region
- Retrieves unique numeric identifier for each thread
- Useful in program logic

# HELLO2

**In the RUN window**

```
$ export OMP_NUM_THREADS=8
$ ./hello2
```

```
[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_hello2.f90

- omp_get_num_threads:
  returns number of active threads

- num_threads and thread_id variables:
  enable useful conditionals

```
    IMPLICIT NONE
```

```
    ! Typically, this is the value of the shell environment variable $OMP_NUM_THREADS,
    ! though users can control the number of threads active via OMP pragmas and functions.
```

```
    num_threads = omp_get_num_threads()
    If (num_threads .gt. 1) Then
        !We might have regions of the code that we only execute if more than one thread is running.

        If (thread_id .eq. 0) Then
            !Sometimes we might like for only a single thread to report certain information.
            !This avoids redundant output.
            Write(thread_string,'(i8)')num_threads
            Write(6,*)"  "//trim(adjustl(thread_string))//" threads are now active."
        Endif

        Write(thread_string,'(i8)')thread_id
        Write(6,*)"  Hello from thread "//trim(adjustl(thread_string))//"."

    Endif
```

```
    !$OMP END PARALLEL

END PROGRAM MAIN
```

# omp_hello2.f90

```
    IMPLICIT NONE
    INTEGER     :: thread_id, num_threads
    CHARACTER*8 :: thread_string


    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

    ! In addition to the thread ID, we can find the total number
    ! of threads active at any given time.  We store this in num_threads.
    ! Typically, this is the value of the shell environment variable $OMP_NUM_THREADS,
    ! though users can control the number of threads active via OMP pragmas and functions.

                                                                                ing.

        Write(thread_string,'(I8)')thread_id
        Write(6,*)"  Hello from thread "//trim(adjustl(thread_string))//"."

    Endif

    !$OMP END PARALLEL

END PROGRAM MAIN
```

**"private" clause**

- Variables are shared among threads unless declared as private
- This can cause unexpected results

**EXERCISE:**

Remove thread_id from the private clause...

$make clean

$make all

$./hello2

# HELLO2

**In the RUN window**

```
$ export OMP_NUM_THREADS=8
$ ./hello2
```

```
[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?**

# HELLO3

## In the RUN window

```
$ export OMP_NUM_THREADS=8
$ ./hello3
```

```
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_hello3.cpp

```
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
    // thread stops at the barrier and resumes once all threads have
    // reached the barrier.

    #pragma omp barrier

    //
    //                                                    ad ID?
    fo
    {

    }
```

**"barrier" directive**

**EXERCISE:**

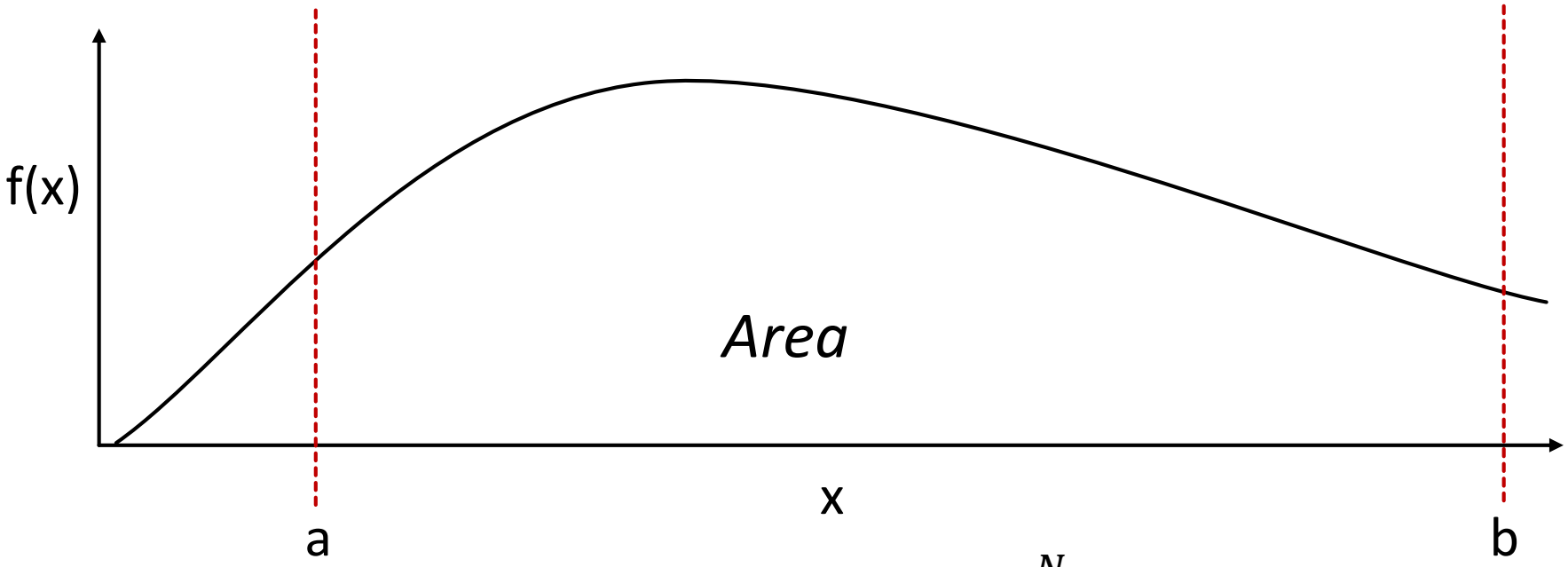
Place a barrier statement within the loop so that threads print their hello statement in ascending order based on thread ID.

$make clean

$make all

$./hello3

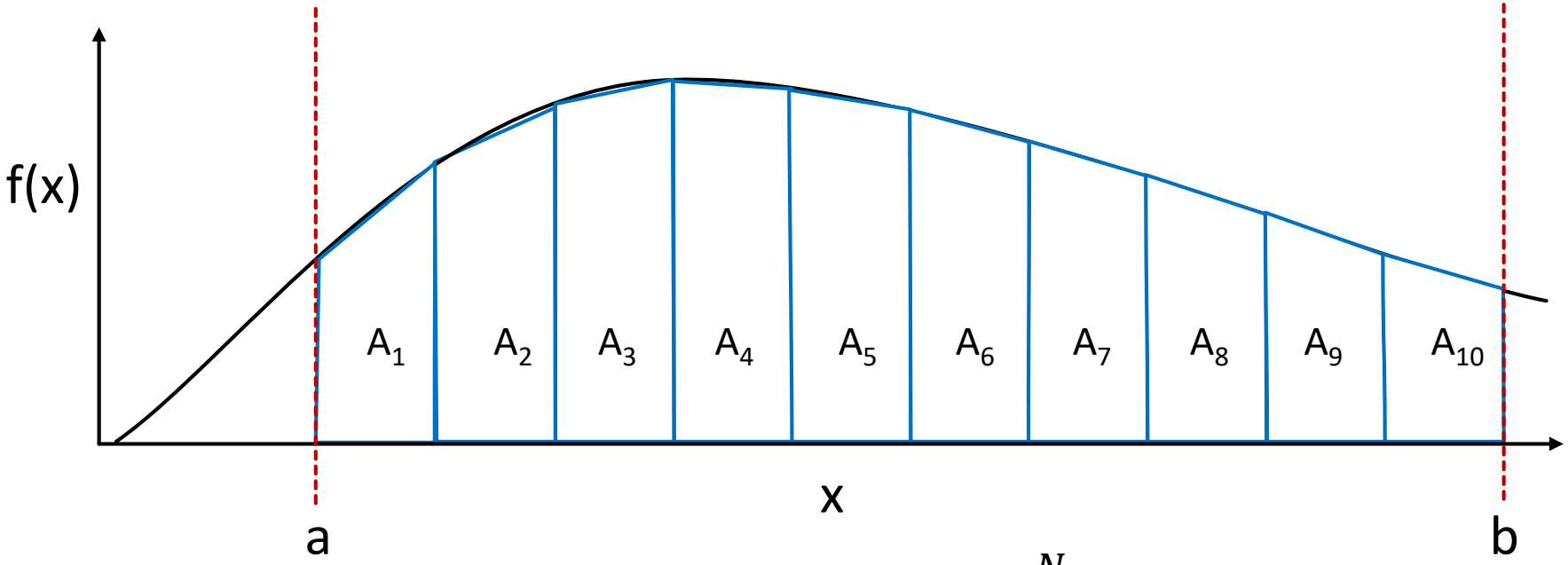
# REAL WORLD PROBLEM: NUMERICAL INTEGRATION




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

I was told there would be no math …

# REAL WORLD PROBLEM: NUMERICAL INTEGRATION




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

# NUMERICAL INTEGRATION

- Open omp_trapezoid.f90 or omp_trapezoid.cpp
- Read through the code
- Compile and run the code:

```
$ make clean
$ make all
$./trapezoid
```

- We are integrating $x^3$ from 1 to 2
- Working, but not parallel yet…

# Examine the loop in the trapezoid_int function

```
!Consider the code below.  Uncomment the
!What variables should we include in the
!!$OMP PARALLEL private()

    ! The OMP DO directive tells the comp
    ! among the active in our parallel re
    ! another thread handles i = 400, etc
    !!$OMP DO
    DO i = 1, ntrap-2
        x = a+i*h
        integral = integral+myfunc(x)
    Enddo
    !!$OMP END DO
!!$OMP END PARALLEL
```

Uncomment these lines

What variables should we include in the private clause?
Hint: what is changing in the loop?

```fortran
!Consider the code below.  Uncomment the
!What variables should we include in the
!$OMP PARALLEL private(i,x,integral)

    ! The OMP DO directive tells the comp
    ! among the active in our parallel re
    ! another thread handles i = 400, etc
    !$OMP DO
    DO i = 1, ntrap-2
        x = a+i*h
        integral = integral+myfunc(x)
    Enddo
    !$OMP END DO
!$OMP END PARALLEL
integral = integral*h
```

Variables that change:
i, x, and integral

<u>NOTE</u>

FORTRAN: !$OMP DO

C++: #pragma omp for

Tells OpenMP to split up the loop work among the available threads

```
!Consider the code below.  Uncomment the
!What variables should we include in the
!$OMP PARALLEL private(i,x,integral)

    ! The OMP DO directive tells the comp
    ! among the active in our parallel re
    ! another thread handles i = 400, etc
    !$OMP DO
    DO i = 1, ntrap-2
        x = a+i*h
        integral = integral+myfunc(x)
    Enddo
    !$OMP END DO
!$OMP END PARALLEL
integral = integral*h
```

Let's try it out!

```
$ make clean
$ make all
$./trapezoid
```

```
[user0038@tutorial-login OpenMP]$ export OMP_NUM_THREADS=2
[user0038@tutorial-login OpenMP]$ ./trapezoid
  Calculating the integral of f(x) = x^3 from 1.000000 to 2.000000
  2 times, using 1000000 trapezoids and 2 threads.
  The integral is 0.000005.
```

Something is off..

# REDUCTION

- Each thread has unique copy of integral
- When the parallel region terminates, we retain only one of those values.
- We want to add them up!
- Make this change:

```
private(i,x,integral)
```

→

```
private(i,x)  reduction(+:integral)
```

- Rerun the code

```
$ make clean
$ make all
$./trapezoid
```

# TIMING

- Once your code is working…
- Change ntests from 2 to 10,000
- Time code for different numbers of threads:

```
$ export OMP_NUM_THREADS=2
$ time ./trapezoid
```

This is how long your run took

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

# III. MPI

# WHAT IS MPI?

- Message Passing Interface
- A library that allows you to run on multiple nodes
- Similar to OpenMP in spirit **but cumbersome**

# WHY SHOULD YOU USE IT?

- You want to run on more than O(10) cores.
- You want *really* to use a supercomputer

# COMPILING WITH MPI

- A different compile command is used for MPI programs

C++

```
$ mpiicpc   mpi_hello1.cpp  –o   hello1
```

FORTRAN

```
$ mpiifort   mpi_hello1.f90  –o   hello1
```

The Makefiles are already set up to do this for you…
Python scripts are not compiled…

# CHOOSE YOUR PATH:

- Within EACH window, cd into appropriate directory:
  - C++ : parallel_programming/C++/MPI
  - FORTRAN : parallel_programming/FORTRAN/MPI
  - Python : parallel_programming/MPI

```
$ module unload all
$ module load intel
$ module load impi
$ module load python/2.7.11
```

# MPI: HELLO WORLD

**1. In the COMPILE window**

```
$ make clean
$ make all
```

**2. In the RUN window**

```
$ mpirun –np 2 ./hello1
OR
$ mpirun –np 2 python hello1.py
```

**3. In the RUN window**

```
$ mpirun –np 5 ./hello1
OR
$ mpirun –np 5 python ./hello1
```

**4. In the COMPILE window**

```
$ nano mpi_hello1.cpp
$ nano mpi_hello1.f90
$ nano hello1.py
```

**mpirun:** parallel execution command for MPI programs

**-np # :** indicates that hello1 should be run with # cores

# MPI_HELLO1.F90

```
PROGRAM MAIN
    Implicit None
    include "mpif.h"  !We always include mpif.h when running MPI programs in fortran
    Character*8 :: rank_string, nproc_string
    Character(MPI_MAX_PROCESSOR_NAME) :: node_name
    Integer :: ierr, n

    !This entire code
    !There is no forki

    !Initialize MPI  (
    Call MPI_INIT( ier

    !Find number of MP
    Call MPI_Comm_size

    !Find this process's rank (my_rank; unique numeric identifier for each MPI process)
    Call MPI_Comm_rank(MPI_COMM_WORLD, my_rank,ierr)

    ! Find the name of this node (node_name)
    Call MPI_Get_processor_name(node_name, name_len,ierr)


    Write(rank_string,'(i8)')my_rank
    Write(nproc_string,'(i8)')num_proc
    Write(6,*)"  Hello from node "//trim(node_name)//", rank "// &
        & trim(adjustl(rank_string))//" out of "//trim(adjustl(nproc_string))//" processes."

    !Terminate communication between processes.
    !No further MPI commands may be executed following finalization.
    Call MPI_Finalize(ierr)
END PROGRAM MAIN
```

**Include statement**

This gives the program access to all the capitalized MPI_XXXX variables you see in the code.

# MPI_HELLO1.F90

```
PROGRAM MAIN
    Implicit None
    include "mpif.h"  !We always include mpif.h when running MPI programs in fortran
    Character*8 :: rank_string, nproc_string
    Character(MPI_MAX_PROCESSOR_NAME) :: node_name
    Integer :: ierr, num_proc, my_rank, name_len

    !This entire code is executed in parallel.
    !There is no forking as with OpenMP.

    !Initialize MPI  (ierr is an error flag with nonzero value if there's a problem)
    Call MPI_INIT( ierr )

    !Find number of MPI processes executing this program (num_proc)
    Call MPI_Comm_size(

    !Find this process'
    Call MPI_Comm_rank(

    ! Find the name of
    Call MPI_Get_proces


    Write(rank_string,
    Write(nproc_string,
    Write(6,*)"  Hello
        & trim(adjustl(

    !Terminate communication between processes.
    !No further MPI commands may be executed following finalization.
    Call MPI_Finalize(ierr)
END PROGRAM MAIN
```

**MPI_INIT**

Initialize inter-process communication. Code is running on all processes, but they are not communicating until this is called.

# MPI_HELLO1.F90

```
PROGRAM MAIN
    Implicit None
    include "mpif.h"  !We always include mpif.h when running MPI programs in fortran
    Character*8 :: rank_string, nproc_string
    Character(MPI_MAX_PROCESSOR_NAME) :: node_name
    Integer :: ierr, num_proc, my_rank, name_len

    !This entire code is executed in parallel.
    !There is no forking as with OpenMP.

    !Initialize MPI  (ierr is an error flag with nonzero value if there's a problem)
    Call MPI_INIT( ierr )

    !Find number of MPI processes executing this program (num_proc)
    Call MPI_Comm_size(

    !Find this process'
    Call MPI_Comm_rank(

    ! Find the name of
    Call MPI_Get_proces


    Write(rank_string,
    Write(nproc_string,
    Write(6,*)"  Hello from node "//trim(node_name)//", rank "// &
        & trim(adjustl(rank_string))//" out of "//trim(adjustl(nproc_string))//" processes."

    !Terminate communication between processes.
    !No further MPI commands may be executed following finalization.
    Call MPI_Finalize(ierr)
END PROGRAM MAIN
```

**NOTE**

When you execute mpirun –np X,
X copies of the code are running.
EVEN IF YOU NEVER CALL MPI_INIT

# MPI_HELLO1.F90

```
PROGRAM MAIN
    Implicit None
    include "mpif.h"  !We always include mpif.h when running MPI programs in fortran
    Character*8 :: rank_string, nproc_string
    Character(MPI_MAX_PROCESSOR_NAME) :: node_name
    Integer :: ierr, num_proc, my_rank, name_len

    !This entire code is executed in parallel.
    !There is no forking as with OpenMP.

    !Initialize MPI  (ierr is an error flag with nonzero value if there's a problem)
    Call MPI_INI

    !Find number
    Call MPI_Com

    !Find this p
    Call MPI_Com

    ! Find the name of this node (node_name)
    Call MPI_Get_processor_name(node_name, name_len,ierr)


    Write(rank_string,'(i8)')my_rank
    Write(nproc_string,'(i8)')num_proc
    Write(6,*)"  Hello from node "//trim(node_name)//", rank "// &
        & trim(adjustl(rank_string))//" out of "//trim(adjustl(nproc_string))//" processes."

    !Terminate communication between processes.
    !No further MPI commands may be executed following finalization.
    Call MPI_Finalize(ierr)
END PROGRAM MAIN
```

**MPI_FINALIZE**

Terminates communication permanently for duration of program

# MPI_HELLO1.F90

```
PROGRAM MAIN
    Implicit None
    include "mpif.h"  !We always include mpif.h when running MPI programs in fortran
```

**MPI_Comm_Size**

Retrieve the number of active MPI ranks (processes) .

Should be X from “mpirun –X”

```
    Call MPI_INIT( ierr )

    !Find number of MPI processes executing this program (num_proc)
    Call MPI_Comm_size(MPI_COMM_WORLD, num_proc,ierr)

    !Find this process's rank (my_rank; unique numeric identifier for each MPI process)
    Call MPI_Comm_rank(MPI_COMM_WORLD, my_rank,ierr)

    ! Find the name of this node (node_name)
    Call MPI_Get_processor_name(node_name, name_len,ierr)


    Write(rank_string,'(i8)')my_rank
    Write(nproc_string,'(i8)')num_proc
    Write(6,*)"  Hello from node "//trim(node_name)//", rank "// &
        & trim(adjustl(rank_string))//" out of "//trim(adjustl(nproc_string))//" processes."

    !Terminate communication between processes.
    !No further MPI commands may be executed following finalization.
    Call MPI_Finalize(ierr)
END PROGRAM MAIN
```

# MPI_HELLO1.F90

```
PROGRAM MAIN
    Implicit None
```

**MPI_Comm_Rank**

Retrieve the numeric ID of this MPI rank.

If you run with “mpirun –X,” this will be anything From 0 to (X-1)

```
    !Find number of MPI processes executing this program (num_proc)
    Call MPI_Comm_size(MPI_COMM_WORLD, num_proc,ierr)

    !Find this process's rank (my_rank; unique numeric identifier for each MPI process)
    Call MPI_Comm_rank(MPI_COMM_WORLD, my_rank,ierr)

    ! Find the name of this node (node_name)
    Call MPI_Get_processor_name(node_name, name_len,ierr)


    Write(rank_string,'(i8)')my_rank
    Write(nproc_string,'(i8)')num_proc
    Write(6,*)"  Hello from node "//trim(node_name)//", rank "// &
        & trim(adjustl(rank_string))//" out of "//trim(adjustl(nproc_string))//" processes."

    !Terminate communication between processes.
    !No further MPI commands may be executed following finalization.
    Call MPI_Finalize(ierr)
END PROGRAM MAIN
```

# MPI_HELLO1.F90

```
PROGRAM MAIN
    Implicit None
    include "mpif.h"  !We always include mpif.h when running MPI programs in fortran
```

**MPI_COMM_WORLD**

**VERY SPECIAL VARIABLE**

An integer identifier for the process pool of all MPI processes in this session.

```
    !Find number of MPI processes executing this program (num_proc)
    Call MPI_Comm_size(MPI_COMM_WORLD, num_proc,ierr)

    !Find this process's rank (my_rank; unique numeric identifier for each MPI process)
    Call MPI_Comm_rank(MPI_COMM_WORLD, my_rank,ierr)

    ! Find the name of this node (node_name)
    Call MPI_Get_processor_name(node_name, name_len,ierr)


    Write(rank_string,'(i8)')my_rank
    Write(nproc_string,'(i8)')num_proc
    Write(6,*)"  Hello from node "//trim(node_name)//", rank "// &
        & trim(adjustl(rank_string))//" out of "//trim(adjustl(nproc_string))//" processes."

    !Terminate communication between processes.
    !No further MPI commands may be executed following finalization.
    Call MPI_Finalize(ierr)
END PROGRAM MAIN
```

C++ and FORTRAN look similar.

Python is a bit different.

# HELLO1.PY

```
#!/usr/bin/env python
def main():
    """
    Parallel Hello World
    """

    from mpi4py import MPI  #importing MPI initializes the MPI library (s
    import sys

    # The program
    num_proc  = MP
    my_rank   = MP                                                    an
    node_name = MP

    sys.stdout.wri
        "  Hello t
        % (node_name, my_rank, num_proc))
    # Once we're finished, we call Disconnect.
    # No further calls to MPI can be made once MPI_Finalize is invoked.
    MPI.COMM_WORLD.Disconnect()
main()
```

**from Mpi4py import MPI**

Initialize inter-process communication.

Analagous to MPI_INIT in C/FORTRAN

# HELLO1.PY

```
#!/usr/bin/env python
def main():
    """
    Parallel Hello World
    """

    from mpi4py import MPI  #importing MPI initializes the MPI library (s
    import sys

    # The program is now running parallel
```

**MPI_COMM_WORLD.Disconnect**

Analagous to MPI_Finalized. Terminates communication.

```
        % (node_name, my_rank, num_proc))
    # Once we're finished, we call Disconnect.
    # No further calls to MPI can be made once MPI_Finalize is invoked.
    MPI.COMM_WORLD.Disconnect()
main()
```

# HELLO3

- The MPI examples are similar to the OpenMP examples.
- MPI also has barrier functionality.
- Run hello3 with 16 MPI ranks. Similar issue as before…

**1. In the COMPILE window**

```
$ make clean
$ make all
```

**2. In the RUN window**

```
$ mpirun –np 16 ./hello3
OR
$ mpirun –np 16 python hello3.py
```

- Examine the mpi_hello3.f90 etc. file
- Replicate the usage of the barrier function so that processes print hello in ascending order based on rank

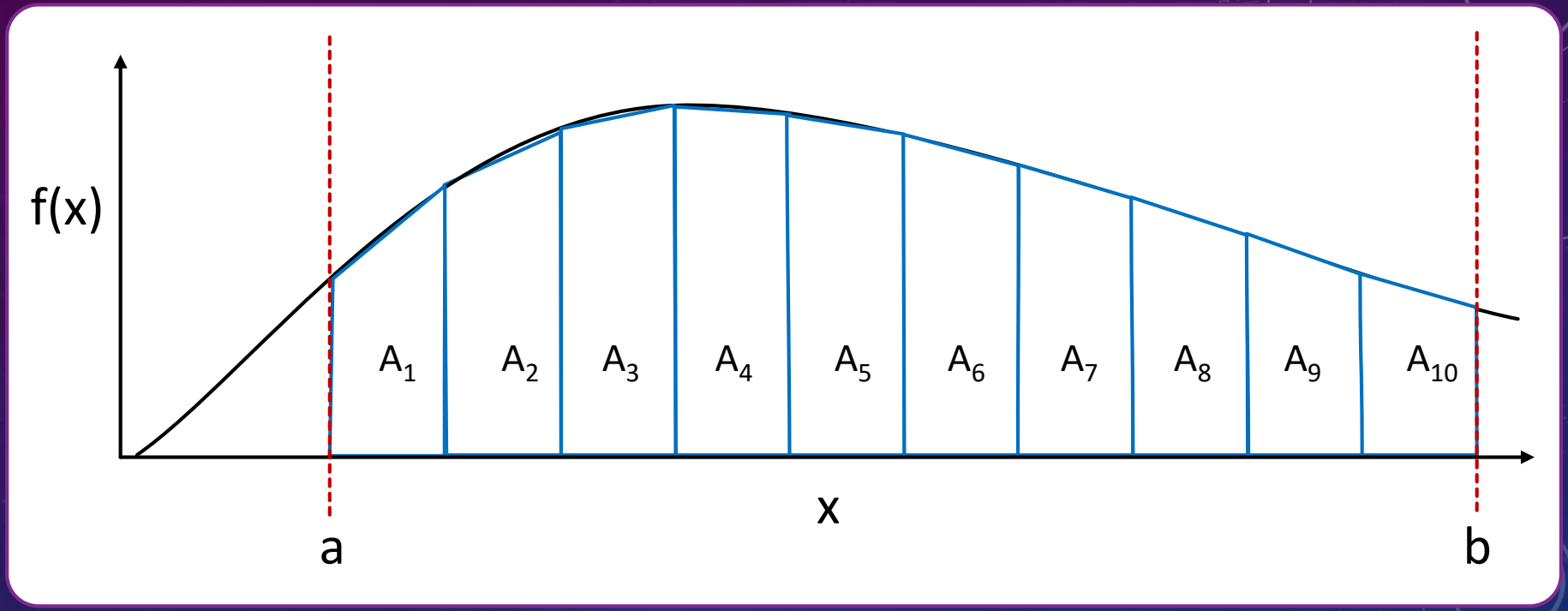
# NUMERICAL INTEGRATION

- Open the trapezoid source file…
- Note the use of allreduce

```
for (int i=0; i<ntests; ++i)
{
    global_integral = 0.0;
    local_integral = trapezoid_int(myxone,myxtwo,ntrap);

    // The call to MPI_Allreduce will sum the value of local_integral across
    // all processes, and store it in global_integral
    MPI_Allreduce(&local_integral, &global_integral, 1, MPI_DOUBLE, MPI_SUM,
                  MPI_COMM_WORLD);
}
```

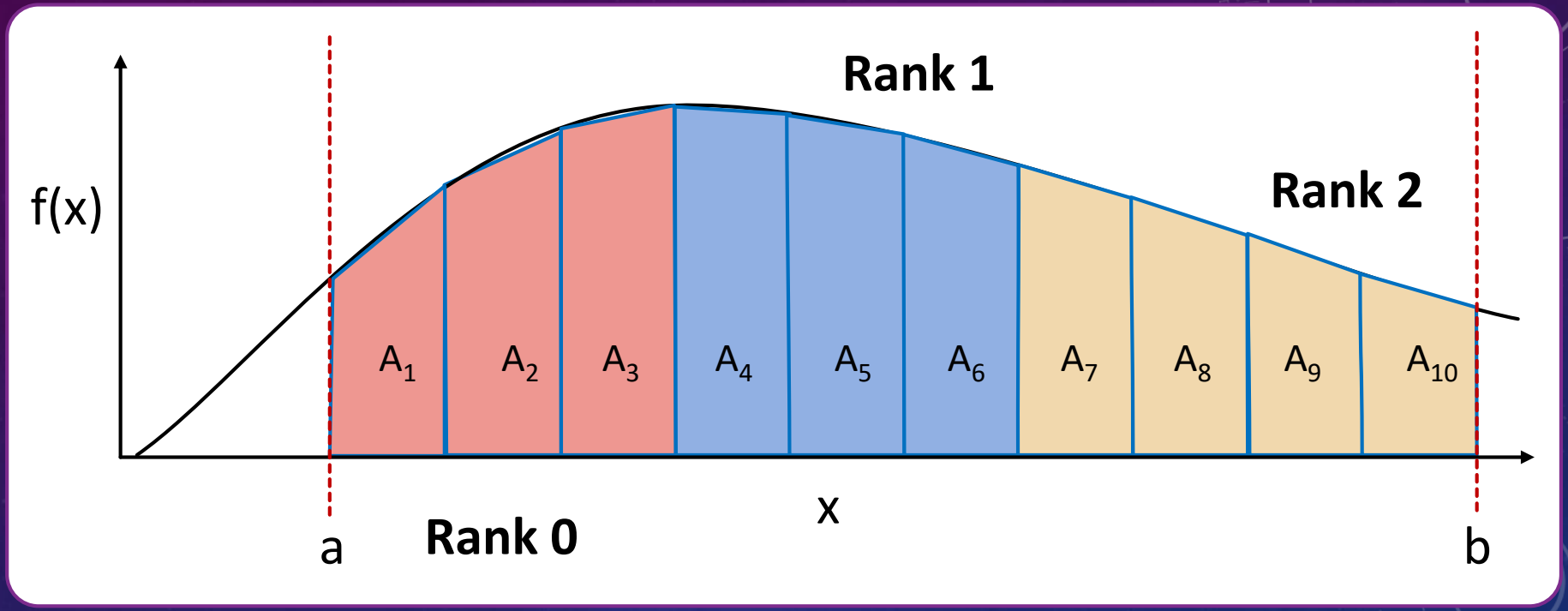
- All processes in the MPI_COMM_WORLD pool add up (MPI_SUM) their value of local integral.
- Result stored in global_integral

# NOT QUITE FINISHED




- OpenMP handles the workload distribution for you
- MPI does not

# PROBLEM DECOMPOSITION




- Idea: Assign each MPI rank a different range in x

# PROBLEM DECOMPOSITION

- We've started the problem setup already:

```
//ntests = 10000; //uncomment
ntrap = 1000000/num_proc;
```

- New local limits of integration are myxone, myxtwo:
- Use my_rank and num_proc to modify these limits appropriately…

```
xone = 1.0;
xtwo = 2.0;
// Each rank should integrate
// What should deltax and myxo

deltax = (xtwo-xone);
myxone = xone;
myxtwo = myxone+deltax;
```

… then run the code!

# MPI: POINT-TO-POINT MESSAGING

- So far, communication among all MPI ranks.
- We can also transmit messages between pairs of ranks.
- Transmission via MPI_SEND & MPI_RECV
- Each message exchange needs:
  - Name of variable to send
  - The rank of the sender and receiver
  - An integer message tag (must match between sender and receiver)
  - The MPI datatype (Integer, real, etc.)
  - The number of values to transmit

# MPI_MESSAGES.F90

```
    If (my_rank .eq. (num_proc-1)) Then
        ! The highest rank sends a message to rank 0
        Write(rank_string2,'(i8)') 0
        Write(6,*)'Rank '//trim(adjustl(rank_string))//' sending token to rank ' &
            & //trim(adjustl(rank_string2))//'.'

        dest = 0  ! The destination rank
        tag = my_rank ! A unique tag for this message (destination should expect same tag)
        nvals = 1 ! The number of values we are transmitting

        !We send the value of the token variable
        Call MPI_Send(token, nvals, MPI_INTEGER, dest,tag, &
            & MPI_COMM_WORLD,ierr)
    Endif

    If (my_rank .eq. 0) Then
        !Rank 0 receives a message from rank num_proc-1
        Write(rank_string2,'(i8)') num_proc-1

        source = num_proc-1  ! The source of the message
        tag = num_proc-1     ! The message tag (must match the tag  set by sender)
        nvals = 1            ! The number of values we will receive

        !The value of token will be overwritten by whatever num_proc-1 sends.
        Call MPI_Recv(token, nvals, MPI_INTEGER, source, tag, &
            & MPI_COMM_WORLD, MPI_STATUS_IGNORE,ierr)
        Write(6,*)'Rank '//trim(adjustl(rank_string))//' has received token from rank ' &
            & //trim(adjustl(rank_string2))//'.'
        Write(6,*)'Token value: ', token
    Endif

    Call MPI_Finalize(ierr)
END PROGRAM MAIN
```

# MPI: SENDING AND RECEIVING

- The sends/receives here are locally blocking
- i.e. they act like a Barrier for the rank that calls them until the send/receive is complete
- Caution!

## Good Code

```
If (my_rank == 0):
        send( to rank N)
        receive( from rank N)
If (my_rank == N):
        receive( from rank 0)
        send( to rank 0)
```

## Bad Code

Rank 0 waits forever

```
If (my_rank == 0):
        send( to rank N)
        receive( from rank N-1)
If (my_rank == N):
        receive( from rank 0)
        send( to rank 0)
```

# EXERCISE: TOKEN PASS

- Modify the logic of mpi_messages.f90
- Have rank 0 send to rank 1.
- Have rank 1 receive token from rank 0, print “received,” and then send token to rank 2 etc.
- Rank N-1 prints “complete” after receiving from rank N-2

  …and program exits.