Course Outline

Parallelism & MPI (10 am - noon)

I. Parallelism
II. Supercomputer Architecture
III. Basic MPI
   (Interlude 1: Computing Pi in parallel)
IV. MPI Collectives
   (Interlude 2: Computing Pi using parallel collectives)

OpenMP & Hybrid Programming (1 - 3 pm)
Course Outline

Parallelism & MPI (10 am - noon)

OpenMP & Hybrid Programming (1 - 3 pm)

I. About OpenMP
II. OpenMP Directives
III. Data Scope
IV. Runtime Library Routines & Environment
V. Using OpenMP
   (Interlude 3: Computing Pi with OpenMP)

VI. Hybrid Programming
    (Interlude 4: Computing Pi with Hybrid Programming)
Parallelism & MPI
I. PARALLELISM

“Parallel Worlds” by alosbennett from
http://www.flickr.com/photos/aloshbennett/3209564747/sizes/l/in/photostream/
I. Parallelism

● Concepts of parallelization
● Serial vs. parallel
● Parallelization strategies
Parallelization Concepts

● When performing task, some subtasks depend on one another, while others do not
● Example: Preparing dinner
  ○ Salad prep independent of lasagna baking
  ○ Lasagna must be assembled before baking
● Likewise, in solving scientific problems, some tasks independent of one another
Serial vs. Parallel

- **Serial**: tasks must be performed in sequence
- **Parallel**: tasks can be performed independently in any order
Serial vs. Parallel: Example

- Preparing lasagna dinner
- **Serial tasks**: making sauce, assembling lasagna, baking lasagna; washing lettuce, cutting vegetables, assembling salad
- **Parallel tasks**: making lasagna, making salad, setting table
Serial vs. Parallel: Graph

Make Sauce
- Cook Noodles
  - Grate Cheese
  - Assemble
    - Bake
      - Lasagna

Wash lettuce
- Wash veg
- Cut lettuce
- Cut veg
- Assemble
  - Salad

Serve Dinner

Prep butter
- Cut bread
  - Spread
    - Bake
      - Garlic Bread

Timeline:
- 4:15
- 4:30
- 5:00
- 5:30
- 6:00
Serial vs. Parallel: Graph
Serial vs. Parallel: Example

- Could have several chefs, each performing one parallel task
- This is concept behind parallel computing
Discussion: Jigsaw Puzzle*

- Suppose we want to do a large, $N$-piece jigsaw puzzle (e.g., $N = 10,000$ pieces)
- Time for one person to complete puzzle: $T$ hours
- How can we decrease walltime to completion?
Discussion: Jigsaw Puzzle

- Impact of having multiple people at the table
  - Walltime to completion
  - Communication
  - Resource contention
- Let number of people = \( p \)
  - Think about what happens when \( p = 1, 2, 4, \ldots, 5000 \)
Discussion: Jigsaw Puzzle

Alternate setup: $p$ people, each at separate table with $N/p$ pieces each

- What is the impact on
  - Walltime to completion
  - Communication
  - Resource contention?
Discussion: Jigsaw Puzzle

Alternate setup: divide puzzle by features, each person works on one, e.g., mountain, sky, stream, tree, meadow, etc.

- What is the impact on
  - Walltime to completion
  - Communication
  - Resource contention?
Parallel Algorithm Design: PCAM

- **Partition**: Decompose problem into fine-grained tasks to maximize potential parallelism
- **Communication**: Determine communication pattern among tasks
- **Agglomeration**: Combine into coarser-grained tasks, if necessary, to reduce communication requirements or other costs
- **Mapping**: Assign tasks to processors, subject to tradeoff between communication cost and concurrency

(from Heath: *Parallel Numerical Algorithms*)
II. ARCHITECTURE

II. Supercomputer Architecture

- What is a supercomputer?
- Conceptual overview of architecture

Cray 1 (1976)

IBM Blue Gene (2005)

Cray XT5 (2009)
What Is a Supercomputer?

- “The biggest, fastest computer right this minute.” – Henry Neeman
- Generally at least 100 times more powerful than PC
- This field of study known as supercomputing, high-performance computing (HPC), or scientific computing
- Scientists use really big computers to solve really hard problems
SMP Architecture

- Massive memory, shared by multiple processors
- Any processor can work on any task, no matter its location in memory
- Ideal for parallelization of sums, loops, etc.
Cluster Architecture

- CPUs on racks, do computations (fast)
- Communicate through networked connections (slow)
- Want to write programs that divide computations evenly but minimize communication
State-of-the-Art Architectures

Today, hybrid architectures pervasive

- Multiple \{8, 12, 16, 24, 32, 68\}-core nodes, connected to other nodes by (slow) interconnect
- Cores in node share memory (like small SMP machines)
- Machine appears to follow cluster architecture (with multi-core nodes rather than single processors)
- To take advantage of all parallelism, use MPI (cluster) and OpenMP (SMP) hybrid programming
State-of-the-Art Architectures

- Hybrid CPU/GPGPU architectures broadly accepted
  - Nodes consist of one (or more) multicore CPU + one (or more) GPU
  - Heavy computations offloaded to GPGPUs
  - Separate memory for CPU and GPU
  - Complicated programming paradigm, outside the scope of today’s training
III. BASIC MPI

“MPI Adventure” by Stefan Jürgensen, from
http://www.flickr.com/photos/94039982@N00/6177616380/sizes/l/in/photostream/
III. Basic MPI

- Introduction to MPI
- Parallel programming concepts
- The Six Necessary MPI Commands
- Example program
Introduction to MPI

- Stands for Message Passing Interface
- Industry standard for parallel programming (200+ page document)
- MPI implemented by many vendors; open source implementations available too
  - Cray, IBM, HPE vendor implementations
  - MPICH, LAM-MPI, OpenMPI (open source)
- MPI function library is used in writing C, C++, or Fortran programs in HPC
Introduction to MPI

- MPI-1 vs. MPI-2: MPI-2 has additional advanced functionality and C++ bindings, but everything learned in this section applies to both standards
- MPI-3: Major revisions (e.g., nonblocking collectives, extensions to one-sided operations), released September 2012, 800+ pages
  - MPI-3.1 released June 2015
  - MPI-3 additions to standard will not be covered today
- MPI-4: Standard currently in development
Parallelization Concepts

- Two primary programming paradigms:
  - **SPMD** (single program, multiple data)
  - **MPMD** (multiple programs, multiple data)
- MPI can be used for either paradigm
SPMD vs. MPMD

● **SPMD**: Write single program that will perform same operation on multiple sets of data
  ○ Multiple chefs baking many lasagnas
  ○ Rendering different frames of movie

● **MPMD**: Write different programs to perform different operations on multiple sets of data
  ○ Multiple chefs preparing four-course dinner
  ○ Rendering different parts of movie frame

● Can also write hybrid program in which some processes perform same task
The Six Necessary MPI Commands

int MPI_Init(int *argc, char **argv)
int MPI_Finalize(void)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
Initiation and Termination

- **MPI_Init(int *argc, char **argv)** initiates MPI
  - Place in body of code after variable declarations and before any MPI commands
- **MPI_Finalize(void)** shuts down MPI
  - Place near end of code, after last MPI command
Environmental Inquiry

- **MPI_Comm_size(MPI_Comm comm, int *size)**
  - Find out number of processes
  - Allows flexibility in number of processes used in program

- **MPI_Comm_rank(MPI_Comm comm, int *rank)**
  - Find out identifier of current process
  - $0 \leq \text{rank} \leq \text{size}-1$
Message Passing: Send

- **MPI_Send**(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - Send message of length *count* items and datatype *datatype* contained in *buf* with tag *tag* to process number *dest* in communicator *comm*
  - E.g., **MPI_Send**(x, 1, MPI_DOUBLE, manager, me, MPI_COMM_WORLD)
Message Passing: Receive

- `MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- Receive message of length `count` items and datatype `datatype` with tag `tag` in buffer `buf` from process number `source` in communicator `comm`, and record status `status`
- E.g. `MPI_Recv(&x, 1, MPI_DOUBLE, source, source, MPI_COMM_WORLD, &status)`
Message Passing

- WARNING! Both standard send and receive functions are blocking
- MPI_Recv returns only after receive buffer contains requested message
- MPI_Send may or may not block until message received (usually blocks)
- Must watch out for deadlock


```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    printf("Sent %d to proc %d, received %d from proc %d\n", me, sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```
Deadlocking Example (Sometimes)

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
    printf("Sent %d to proc %d, received %d from proc %d\n", me, sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```
Deadlocking Example (Safe)

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    if (me%2 == 0) {
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
    } else {
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    }
    printf("Sent %d to proc %d, received %d from proc %d\n", me, sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```
Explaination: Always Deadlocking Example

- Logically incorrect
- Deadlock caused by blocking `MPI_Recvs`
- All processes wait for corresponding `MPI_Sends` to begin, which never happens
Explanation: Sometimes Deadlocking Example

- Logically correct
- Deadlock could be caused by `MPI_Sends` competing for buffer space
- Unsafe because depends on system resources
- Solutions:
  - Reorder sends and receives, like safe example, having evens send first and odds send second
  - Use non-blocking sends and receives or other advanced functions from MPI library (see MPI standard for details)
INTERLUDE 1: COMPUTING PI IN PARALLEL

“Pi of Pi” by spellbee2, from
http://www.flickr.com/photos/49825386@N08/7253578340/sizes/l/in/photostream/
Interlude 1: Computing $\pi$ in Parallel

- Project Description
- Serial Code
- Parallelization Strategies
- Your Assignment
Project Description

- We want to compute $\pi$
- One method: method of darts*
- Ratio of area of square to area of inscribed circle proportional to $\pi$

* This is a TERRIBLE way to compute pi! Don’t do this in real life!!!! (See Appendix 1 for better ways)

Method of Darts

- Imagine dartboard with circle of radius $R$ inscribed in square
- Area of circle $= \pi R^2$
- Area of square $= (2R)^2 = 4R^2$
- Area of circle
  Area of square $= \frac{\pi R^2}{4R^2} = \frac{\pi}{4}$

Method of Darts

● Ratio of areas proportional to $\pi$
● How to find areas?
  ○ Suppose we threw darts (completely randomly) at dartboard
  ○ Count # darts landing in circle & total # darts landing in square
  ○ Ratio of these numbers gives approximation to ratio of areas
  ○ Quality of approximation increases with # darts thrown
Method of Darts

\[ \pi = 4 \times \frac{\text{# darts inside circle}}{\text{# darts thrown}} \]

Method of Darts cake in celebration of Pi Day 2009, Rebecca Hartman-Baker
Method of Darts

- Okay, Rebecca, but how in the world do we simulate this experiment on a computer?
- Decide on length $R$
- Generate pairs of random numbers $(x, y)$ s.t.
  $$-R \leq (x, y) \leq R$$
- If $(x, y)$ within circle (i.e., if $(x^2 + y^2) \leq R^2$) add one to tally for inside circle
- Lastly, find ratio
#include "lcgenerator.h"

static long num_trials = 1000000;

int main() {
    long i;
    long Ncirc = 0;
    double pi, x, y;
    double r = 1.0; // radius of circle
    double r2 = r*r;

    for (i = 0; i < num_trials; i++) {
        x = r*lcgrandom();
        y = r*lcgrandom();
        if ((x*x + y*y) <= r2)
            Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc)/((double)num_trials);
    printf("\n For %ld trials, pi = %f\n", num_trials, pi);
    return 0;
}
/// Random number generator -- and not a very good one, either!

static long MULTIPLIER = 1366;
static long ADDEND = 150889;
static long PMOD = 714025;
long random_last = 0;

// This is not a thread-safe random number generator

double lcgrandom() {
    long random_next;
    random_next = (MULTIPLIER * random_last + ADDEND)%PMOD;
    random_last = random_next;

    return ((double)random_next/(double)PMOD);
}
! First, the pseudorandom number generator

real function lcgrandom()
    integer*8, parameter :: MULTIPLIER = 1366
    integer*8, parameter :: ADDEND = 150889
    integer*8, parameter :: PMOD = 714025
    integer*8, save :: random_last = 0

    integer*8 :: random_next = 0
    random_next = mod((MULTIPLIER * random_last + ADDEND), PMOD)
    random_last = random_next
    lcgrandom = (1.0*random_next)/PMOD
    return
end
! Now, we compute pi
program darts
    implicit none
    integer*8 :: num_trials = 1000000, i = 0, Ncirc = 0
    real :: pi = 0.0, x = 0.0, y = 0.0, r = 1.0
    real :: r2 = 0.0
    real :: lcgrandom
    r2 = r*r
    do i = 1, num_trials
        x = r*lcgrandom()
        y = r*lcgrandom()
        if ((x*x + y*y) .le. r2) then
            Ncirc = Ncirc+1
        end if
    end do
    pi = 4.0*((1.0*Ncirc)/(1.0*num_trials))
    print*, ' For ', num_trials, ' trials, pi = ', pi
end
Parallelization Strategies

- What tasks independent of each other?
- What tasks must be performed sequentially?
- Using PCAM parallel algorithm design strategy
“Decompose problem into fine-grained tasks to maximize potential parallelism”

Finest grained task: throw of one dart
Each throw independent of all others
If we had huge computer, could assign one throw to each processor
Communication

“Determine communication pattern among tasks”

- Each processor throws dart(s) then sends results back to manager process
Agglomeration

“Combine into coarser-grained tasks, if necessary, to reduce communication requirements or other costs”

- To get good value of \( \pi \), must use millions of darts
- We don’t have millions of processors available
- Furthermore, communication between manager and millions of worker processors would be very expensive
- Solution: divide up number of dart throws evenly between processors, so each processor does a share of work
Mapping

“Assign tasks to processors, subject to tradeoff between communication cost and concurrency”

- Assign role of “manager” to processor 0
- Processor 0 will receive tallies from all the other processors, and will compute final value of $\pi$
- Every processor, including manager, will perform equal share of dart throws
Your Assignment

- Clone the whole assignment (including answers!) to Cori from the repository with: `git clone https://github.com/hartmanbaker/Developing-with-MPI-and-OpenMP.git`
- Copy `darts.c/lcgenerator.h` or `darts.f` (your choice) from `Developing-with-MPI-and-OpenMP/darts-suite/{c, fortran}`
- Parallelize the code using the 6 basic MPI commands
- Rename your new MPI code `darts-mpi.c` or `darts-mpi.f`
IV. MPI COLLECTIVES

MPI Collectives

- Communication involving group of processes
- Collective operations
  - Broadcast
  - Gather
  - Scatter
  - Reduce
  - All-
  - Barrier
Broadcast

- Perhaps one message needs to be sent from manager to all worker processes
- Could send individual messages
- Instead, use broadcast – more efficient, faster
- int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
Gather

- All processes need to send same (similar) message to manager
- Could implement with each process calling `MPI_Send(...)` and manager looping through `MPI_Recv(...)`
- Instead, use gather operation – more efficient, faster
- Messages concatenated in rank order
- `int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
- Note: `recvcount` = # items received from each process, not total
Gather

- Maybe some processes need to send longer messages than others
- Allow varying data count from each process with MPI_Gatherv(...)  
  ```c
  int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)
  ```
- `recvcounts` is array; entry `i` in `displs` array specifies displacement relative to `recvbuf[0]` at which to place data from corresponding process number
Scatter

- Inverse of gather: split message into \( NP \) equal pieces, with \( i \)th segment sent to \( i \)th process in group
- \textbf{int MPI\_Scatter(void* sendbuf, int sendcount, MPI\_Datatype sendtype, void* recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPI\_Comm comm)}
- Send messages of varying sizes across processes in group: \textbf{MPI\_Scatterv(…)}
- \textbf{int MPI\_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI\_datatype sendtype, void* recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPI\_Comm comm)}
Reduce

- Perhaps we need to do sum of many subsums owned by all processors
- Perhaps we need to find maximum value of variable across all processors
- Perform global reduce operation across all group members
  
  ```c
  int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  ```
# Reduce: Predefined Operations

<table>
<thead>
<tr>
<th>MPI_Op</th>
<th>Meaning</th>
<th>Allowed Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>Integer, floating point</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
<td>Integer, floating point</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td>Integer, floating point, complex</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
<td>Integer, floating point, complex</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical and</td>
<td>Integer, logical</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise and</td>
<td>Integer, logical</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical or</td>
<td>Integer, logical</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise or</td>
<td>Integer, logical</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical xor</td>
<td>Integer, logical</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise xor</td>
<td>Integer, logical</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum value &amp; location</td>
<td>*</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum value &amp; location</td>
<td>*</td>
</tr>
</tbody>
</table>
Reduce: Operations

- **MPI_MAXLOC and MPI_MINLOC**
  - Returns \{max, min\} and rank of first process with that value
  - Use with special MPI pair datatype arguments:
    - MPI_FLOAT_INT \(\text{float and int}\)
    - MPI_DOUBLE_INT \(\text{double and int}\)
    - MPI_LONG_INT \(\text{long and int}\)
    - MPI_2INT \(\text{pair of int}\)
  - See MPI standard for more details

- **User-defined operations**
  - Use `MPI_Op_create(…)` to create new operations
  - See MPI standard for more details
All- Operations

- Sometimes, may want to have result of gather, scatter, or reduce on all processes
- Gather operations
  - `int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
  - `int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)`
All-to-All Scatter/Gather

- Extension of Allgather in which each process sends distinct data to each receiver
- Block $j$ from process $i$ is received by process $j$ into $i$th block of recvbuf
- \texttt{int MPI\_Alltoall(void* sendbuf, int sendcount, MPI\_Datatype sendtype, void* recvbuf, int recvcount, MPI\_Datatype recvtype, MPI\_Comm comm)}
- Corresponding \texttt{MPI\_Alltoallv} function also available
All-Reduce

- Same as `MPI_Reduce` except result appears on all processes

```c
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```
Barrier

- In algorithm, may need to synchronize processes
- Barrier blocks until all group members have called it
- `int MPI_Barrier(MPI_Comm comm)`
Bibliography/Resources: MPI/MPI Collectives

Bibliography/Resources: MPI/MPI Collectives

- Message Passing Interface (MPI) Tutorial
  [https://computing.llnl.gov/tutorials/mpi/](https://computing.llnl.gov/tutorials/mpi/)

- MPI Standard at MPI Forum
  - MPI 1.1:
  - MPI-2.2:
  - MPI 3.1:
INTERLUDE 2: COMPUTING PI WITH MPI

COLLECTIVES

“Pi-Shaped Power Lines at Fermilab” by Michael Kappel from
http://www.flickr.com/photos/m-i-k-e/4781834200/sizes/l/in/photostream/
Interlude 2: Computing $\pi$ with MPI Collectives

- In previous Interlude, you used the 6 basic MPI routines to develop a parallel program using the Method of Darts to compute $\pi$
- The communications in previous program could be made more efficient by using collectives
- Your assignment: update your MPI code to use collective communications
- Rename it `darts-collective.c` or `darts-collective.f`
OpenMP & Hybrid Programming
Outline

I. About OpenMP
II. OpenMP Directives
III. Data Scope
IV. Runtime Library Routines and Environment Variables
V. Using OpenMP
VI. Hybrid Programming
I. ABOUT OPENMP
About OpenMP

- Industry-standard shared memory programming model
- Developed in 1997
- OpenMP Architecture Review Board (ARB) determines additions and updates to standard
- Current standard: 5.0 (November 2018)
Advantages to OpenMP

- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Can express simple or complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, so code is easy to read
- Single source code for OpenMP and non-OpenMP – non-OpenMP compilers simply ignore OMP directives
OpenMP Programming Model

- Application Programmer Interface (API) is combination of
  - Directives
  - Runtime library routines
  - Environment variables

- API falls into three categories
  - Expression of parallelism (flow control)
  - Data sharing among threads (communication)
  - Synchronization (coordination or interaction)
Parallelism

- Shared memory, thread-based parallelism
- Explicit parallelism (parallel regions)
- Fork/join model

Source: https://computing.llnl.gov/tutorials/openMP/
II. OPENMP DIRECTIVES

II. OpenMP Directives

- Syntax overview
- Parallel
- Loop
- Sections
- Synchronization
- Reduction
Syntax Overview: C/C++

- Basic format
  - `#pragma omp directive-name [clause] newline`
- All directives followed by newline
- Uses pragma construct (pragma = Greek for “thing done”)
- Case sensitive
- Directives follow standard rules for C/C++ compiler directives
- Use curly braces (not on pragma line) to denote scope of directive
- Long directive lines can be continued by escaping newline character with \

NERSC

BERKELEY LAB

U.S. DEPARTMENT OF ENERGY

Office of Science
Syntax Overview: Fortran

● Basic format:
  ○ *sentinel directive-name* [clause]

● Three accepted sentinels: !$omp  *$omp  c$omp

● Some directives paired with end clause

● Fixed-form code:
  ○ Any of three sentinels beginning at column 1
  ○ Initial directive line has space/zero in column 6
  ○ Continuation directive line has non-space/zero in column 6
  ○ Standard rules for fixed-form line length, spaces, etc. apply

● Free-form code:
  ○ !$omp only accepted sentinel
  ○ Sentinel can be in any column, but must be preceded by only white space and followed by a space
  ○ Line to be continued must end in & and following line begins with sentinel
  ○ Standard rules for free-form line length, spaces, etc. apply
OpenMP Directives: Parallel

- A block of code executed by multiple threads
- Syntax:

```c
#pragma omp parallel private(list) shared(list)
{
    /* parallel section */
}

!$omp parallel private(list) &
!$omp shared(list)
! Parallel section
!$omp end parallel
```
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
    int tid;
    printf("Hello world from threads:\n");
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("<%d>\n", tid);
    }
    printf("I am sequential now\n");
    return 0;
}
program hello
integer tid, omp_get_thread_num
write(*,*) 'Hello world from threads:'
 !$omp parallel private(tid)
tid = omp_get_thread_num()
write(*,*) '<', tid, '>
 !$omp end parallel
write(*,*) 'I am sequential now'
end
Simple Example: Output

Output 1
Hello world from threads:
<0>
<1>
<2>
<3>
<4>
I am sequential now

Output 2
Hello world from threads:
<1>
<2>
<0>
<4>
<3>
I am sequential now

Order of execution is scheduled by OS!!!
OpenMP Directives: Loop

- Iterations of the loop following the directive are executed in parallel
- Syntax (C):

```c
#pragma omp for schedule(type [,chunk]) private(list)\ shared(list) nowait
{
    /* for loop */
}
```
OpenMP Directives: Loop

- Syntax (Fortran):
  
  ```fortran
  !$omp do schedule (type [,chunk]) &
  !omp private(list) shared(list)
  C do loop goes here
  !$omp end do nowait
  ```

  - `type = {static, dynamic, guided, runtime}`
  - If `nowait` specified, threads do not synchronize at end of loop
OpenMP Directives: Loop Scheduling

- Default scheduling determined by implementation
- Static
  - ID of thread performing particular iteration is function of iteration number and number of threads
  - Statically assigned at beginning of loop
  - Load imbalance may be issue if iterations have different amounts of work
  - Low overhead
- Dynamic
  - Assignment of threads determined at runtime (round robin)
  - Each thread gets more work after completing current work
  - Load balance is possible
  - Introduces extra overhead
OpenMP Directives: Loop Scheduling

<table>
<thead>
<tr>
<th>Type</th>
<th>Chunks</th>
<th>Chunk Size</th>
<th># Chunks</th>
<th>Overhead</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>N</td>
<td>$N/P$</td>
<td>$P$</td>
<td>Lowest</td>
<td>Simple Static</td>
</tr>
<tr>
<td>static</td>
<td>Y</td>
<td>$C$</td>
<td>$N/C$</td>
<td>Low</td>
<td>Interleaved</td>
</tr>
<tr>
<td>dynamic</td>
<td>N</td>
<td>$N/P$</td>
<td>$P$</td>
<td>Medium</td>
<td>Simple dynamic</td>
</tr>
<tr>
<td>dynamic</td>
<td>Y</td>
<td>$C$</td>
<td>$N/C$</td>
<td>High</td>
<td>Dynamic</td>
</tr>
<tr>
<td>guided</td>
<td>N/A</td>
<td>$\leq N/P$</td>
<td>$\leq N/C$</td>
<td>Highest</td>
<td>Dynamic optimized</td>
</tr>
<tr>
<td>runtime</td>
<td>Varies</td>
<td>Varies</td>
<td>Varies</td>
<td>Varies</td>
<td>Set by environment variable</td>
</tr>
</tbody>
</table>

Note: $N = \text{size of loop}$, $P = \text{number of threads}$, $C = \text{chunk size}$
Which Loops are Parallelizable?

**Parallelizable**
- Number of iterations known upon entry, and does not change
- Each iteration independent of all others
- No data dependence

**Not Parallelizable**
- Conditional loops (many while loops)
- Iterator loops (e.g., iterating over `std::list<...>` in C++)
- Iterations dependent upon each other
- Data dependence

**Trick:** If a loop can be run backwards and get the same results, then it is almost always parallelizable!
Example: Parallelizable?

/* Gaussian Elimination (no pivoting): \( \mathbf{x} = \mathbf{A}\backslash\mathbf{b} \) */

for (int i = 0; i < N-1; i++) {
    for (int j = i; j < N; j++) {
        double ratio = A[j][i]/A[i][i];
        for (int k = i; k < N; k++) {
            A[j][k] -= (ratio*A[i][k]);
            b[j] -= (ratio*b[i]);
        }
    }
}
Example: Parallelizable?
Example: Parallelizable?

- **Outermost Loop (i):**
  - $N-1$ iterations
  - Iterations depend upon each other (values computed at step $i-1$ used in step $i$)

- **Inner loop (j):**
  - $N-i$ iterations (constant for given $i$)
  - Iterations can be performed in any order

- **Innermost loop (k):**
  - $N-i$ iterations (constant for given $i$)
  - Iterations can be performed in any order
Example: Parallelizable?

/* Gaussian Elimination (no pivoting): x = A\b */

for (int i = 0; i < N-1; i++) {
    #pragma omp parallel for
    for (int j = i; j < N; j++) {
        double ratio = A[j][i]/A[i][i];
        for (int k = i; k < N; k++) {
            A[j][k] -= (ratio*A[i][k]);
            b[j] -= (ratio*b[i]);
        }
    }
}

Note: can combine parallel and for into single pragma
OpenMP Directives: Sections

- Non-iterative work-sharing construct
- Divide enclosed sections of code among threads
- Section directives nested within sections directive
- Syntax: C/C++

```c
#pragma omp sections
{
    #pragma omp section
    /* first section */
    #pragma omp section
    /* next section */
}
```

Fortran

```fortran
!$omp sections
```

```c
 First section
```

```fortran
!$omp section
```

```c
 Second section
```

```fortran
!$omp section
```

```c
!$omp end sections
```

```fortran
!$omp end sections
```
Example: Sections

#include <omp.h>
define N 1000
int main () {
    int i;
    double a[N], b[N];
    double c[N], d[N];
    /* Some initializations */
    for (i=0; i < N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }
    #pragma omp parallel shared(a,b,c,d)
    private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N; i++)
                c[i] = a[i] + b[i];
            #pragma omp section
            for (i=0; i < N; i++)
                d[i] = a[i] * b[i];
        } /* end of sections */
    } /* end of parallel section */
    return 0;
}
OpenMP Directives: Synchronization

- Sometimes, need to make sure threads execute regions of code in proper order
  - Maybe one part depends on another part being completed
  - Maybe only one thread need execute a section of code
- Synchronization directives
  - Critical
  - Barrier
  - Single
OpenMP Directives: Synchronization

- Critical
  - Specifies section of code that must be executed by only one thread at a time
  - Syntax: C/C++
    
    ```
    #pragma omp critical (name)
    ```
  - Fortran
    
    ```
    !$omp critical (name)
    !$omp end critical
    ```
  - Names are global identifiers – critical regions with same name are treated as same region
OpenMP Directives: Synchronization

- **Single**
  - Enclosed code is to be executed by only one thread
  - Useful for thread-unsafe sections of code (e.g., I/O)
  - Syntax:
    - C/C++: `#pragma omp single`  
    - Fortran: `!$omp single`  
    - `!$omp end single`
OpenMP Directives: Synchronization

- **Barrier**
  - Synchronizes all threads: thread reaches barrier and waits until all other threads have reached barrier, then resumes executing code following barrier
  - Syntax: C/C++
    - `#pragma omp barrier`
    - `!$OMP barrier`
  - Fortran
  - Sequence of work-sharing and barrier regions encountered must be the same for every thread
OpenMP Directives: Reduction

- Reduces list of variables into one, using operator (e.g., max, sum, product, etc.)
- Syntax

```plaintext
#pragma omp reduction(op : list)
!$omp reduction(op : list)
```

where list is list of variables and op is one of following:

- C/C++: +, -, *, &, ^, |, &&, ||, max, min
- Fortran: +, -, *, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor
III. VARIABLE SCOPE

“M119A2 Scope” by Georgia National Guard, source: http://www.flickr.com/photos/ganatlguard/5934238668/sizes/l/in/photostream/
III. Variable Scope

- About variable scope
- Scoping clauses
- Common mistakes
About Variable Scope

● Variables can be shared or private within a parallel region
● Shared: one copy, shared between all threads
  ○ Single common memory location, accessible by all threads
● Private: each thread makes its own copy
  ○ Private variables exist only in parallel region
About Variable Scope

● By default, all variables shared except
  ○ Index values of parallel region loop – **private by default**
  ○ Local variables and value parameters within subroutines called within parallel region – **private**
  ○ Variables declared within lexical extent of parallel region – **private**

● Variable scope is the most common source of errors in OpenMP codes
  ○ Correctly determining variable scope is key to correctness and performance of your code
Variable Scoping Clauses: Shared

- Shared variables: `shared (list)`
  - By default, all variables shared unless otherwise specified
  - All threads access this variable in same location in memory
  - Race conditions can occur if access is not carefully controlled
Variable Scoping Clauses: Private

- **Private:** `private (list)`
  - Variable exists only within parallel region
  - Value undefined at start and after end of parallel region
- **Private starting with defined values:** `firstprivate (list)`
  - Private variables initialized to be the value held immediately before entry into parallel region
- **Private ending with defined value:** `lastprivate (list)`
  - At end of loop, set variable to value set by final iteration of loop
Common Mistakes

● A variable that should be private is public
  ○ Something unexpectedly gets overwritten
  ○ Solution: explicitly declare all variable scope

● Nondeterministic execution
  ○ Different results from different executions

● Race condition
  ○ Sometimes you get the wrong answer
  ○ Solutions:
    ■ Look for overwriting of shared variable
    ■ Use a tool such as Cray Reveal or Parallelware Analyzer to rescope your loop
Find the Mistake(s)!

/* Gaussian Elimination (no pivoting): x = A\b */
int i, j, k;
double ratio;
for (i = 0; i < N-1; i++) {
    #pragma omp parallel for
    for (j = i; j < N; j++) {
        ratio = A[j][i]/A[i][i];
        for (k = i; k < N; k++) {
            A[j][k] -= (ratio*A[i][k]);
            b[j] -= (ratio*b[i]);
        }
    }
}

j, k, & ratio are shared variables by default. Depending on compiler, j & k may be optimized out & therefore not impact correctness, but ratio will always lead to errors! Depending how loop is scheduled, you will see different answers.
Fix the Mistake(s)!

```c
/* Gaussian Elimination (no pivoting):   x = A\b   */
int i, j, k;
double ratio;
for (i = 0; i < N-1; i++) {
    #pragma omp parallel for private (j, k, ratio) \ 
    shared (A, b, N) default none
    for (j = i; j < N; j++) {
        ratio = A[j][i]/A[i][i];
        for (k = i; k < N; k++) {
            A[j][k] -= (ratio*A[i][k]);
            b[j] -= (ratio*b[i]);
        }
    }
}
```

By setting **default none**, compiler will catch any variables not explicitly scoped.
IV. RUNTIME LIBRARY ROUTINES & ENVIRONMENT VARIABLES

OpenMP Runtime Library Routines

- **void omp_set_num_threads(int num_threads)**
  - Sets number of threads used in next parallel region
  - Must be called from serial portion of code

- **int omp_get_num_threads()**
  - Returns number of threads currently in team executing parallel region from which it is called

- **int omp_get_thread_num()**
  - Returns rank of thread
  - $0 \leq \text{omp_get_thread_num}() < \text{omp_get_num_threads}()$
OpenMP Environment Variables

- Set environment variables to control execution of parallel code
- **OMP_SCHEDULE**
  - Determines how iterations of loops are scheduled
  - E.g., `export OMP_SCHEDULE="dynamic, 4"`
- **OMP_NUM_THREADS**
  - Sets maximum number of threads
  - E.g., `export OMP_NUM_THREADS=4`
V. USING OPENMP
Conditional Compilation

- Can write single source code for use with or without OpenMP
  - Pragmas are ignored if OpenMP disabled
- What about OpenMP runtime library routines?
  - \_OPENMP macro is defined if OpenMP available: can use this to conditionally include omp.h header file, else redefine runtime library routines
Conditional Compilation

```c
#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

... int me = omp_get_thread_num();
...```
Enabling OpenMP

- Most standard compilers support OpenMP directives
- Enable using compiler flags

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Intel</th>
<th>GNU</th>
<th>PGI</th>
<th>Cray</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>-qopenmp</td>
<td>-fopenmp</td>
<td>-mp</td>
<td>-h omp</td>
</tr>
</tbody>
</table>
Running Programs with OpenMP Directives

- Set OpenMP environment variables in batch scripts (e.g., include definition of `OMP_NUM_THREADS` in script)
- Example: to run a code with 8 MPI processes and 4 threads/MPI process on Cori:
  - `export OMP_NUM_THREADS=4`
  - `export OMP_PLACES=threads`
  - `export OMP_PROC_BIND=spread`
  - `srun -n 8 -c 8 --cpu_bind=cores ./myprog`
- Use the NERSC jobsctipt generator for best results: [https://my.nersc.gov/script_generator.php](https://my.nersc.gov/script_generator.php)
INTERLUDE 3: COMPUTING PI WITH OPENMP

Interlude 3: Computing $\pi$ with OpenMP

- Think about the original darts program you downloaded (darts.c/1cgenerator.h or darts.f)
- How could we exploit shared-memory parallelism to compute $\pi$ with the method of darts?
- What possible pitfalls could we encounter?
- Your assignment: parallelize the original darts program using OpenMP
- Rename it darts-omp.c or darts-omp.f
VI. HYBRID PROGRAMMING
VI. Hybrid Programming

- Motivation
- Considerations
- MPI threading support
- Designing hybrid algorithms
- Examples
Motivation

● Multicore architectures are here to stay
  ○ Macro scale: distributed memory architecture, suitable for MPI
  ○ Micro scale: each node contains multiple cores and shared memory, suitable for OpenMP

● Obvious solution: use MPI between nodes, and OpenMP within nodes

● Hybrid programming model
Considerations

○ Sounds great, Rebecca, but is hybrid programming always better?
  ○ No, not always
  ○ Especially if poorly programmed 😊
  ○ Depends also on suitability of architecture

○ Think of accelerator model
  ○ in omp parallel region, use power of multicores; in serial region, use only 1 processor
  ○ If your code can exploit threaded parallelism “a lot”, then try hybrid programming
Considerations

● Hybrid parallel programming model
  ○ Are communication and computation discrete phases of algorithm?
  ○ Can/do communication and computation overlap?

● Communication between threads
  ○ Communicate only outside of parallel regions
  ○ Assign a manager thread responsible for inter-process communication
  ○ Let some threads perform inter-process communication
  ○ Let all threads communicate with other processes
MPI Threading Support

- MPI-2 standard defines four threading support levels
  - (0) MPI_THREAD_SINGLE only one thread allowed
  - (1) MPI_THREAD_FUNNELED master thread is only thread permitted to make MPI calls
  - (2) MPI_THREAD_SERIALIZED all threads can make MPI calls, but only one at a time
  - (3) MPI_THREAD_MULTIPLE no restrictions
  - (0.5) MPI calls not permitted inside parallel regions (returns MPI_THREAD_SINGLE) – this is MPI-1
What Threading Model Does My Machine Support?

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

    printf("Supports level %d of %d %d %d %d\n", provided,
    MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED,
    MPI_THREAD_SERIALIZED, MPI_THREAD_MULTIPLE);

    MPI_Finalize();
    return 0;
}
```
What Threading Model Does My Machine Support?

rjhb@cori03:~/test> cc -o threadmodel threadmodel.c
rjhb@cori03:~/test> salloc -C haswell -q interactive
salloc: Granted job allocation 22559071
salloc: Waiting for resource configuration
salloc: Nodes nid00189 are ready for job
rjhb@nid00189:~/test> srun -n 1 ./threadmodel

Supports level 2 of 0 1 2 3
MPI_Init_thread

- **MPI_Init_thread**(int required, int *supported)
  - Use this instead of **MPI_Init**(…)
  - **required**: the level of thread support you want
  - **supported**: the level of thread support provided by implementation (ideally = **required**, but if not available, returns lowest level > **required**; failing that, largest level < **required**)
  - Using **MPI_Init**(…) is equivalent to **required** = **MPI_THREAD_SINGLE**

- **MPI_Finalize**( ) should be called by same thread that called **MPI_Init_thread**(…)

135
Other Useful MPI Functions

- **MPI_Is_thread_main(int *flag)**
  - Thread calls this to determine whether it is main thread

- **MPI_Query_thread(int *provided)**
  - Thread calls to query level of thread support
Supported Threading Models: Single

- Use single pragma

```c
#pragma omp parallel
{
    #pragma omp barrier
    #pragma omp single
    {
        MPI_Xyz(...);
    }
    #pragma omp barrier
}
```
**Supported Threading Models: Funneled**

- Cray & Intel MPI implementations support funneling
- Use master pragma

```c
#pragma omp parallel
{
   #pragma omp barrier
   #pragma omp master
   {
      MPI_Xyz(...);
   }
   #pragma omp barrier
}
```
Supported Threading Models: Serialized

- Cray & Intel MPI implementations support serialized
- Use single pragma

```c
#pragma omp parallel
{
    #pragma omp barrier
    #pragma omp single
    {
        MPI_Xyz(...);
    }
    //Don't need omp barrier
}
```
Supported Threading Models: Multiple

- Intel MPI implementation supports multiple!
  - (Cray MPI can turn on multiple support with env variables, but performance is sub-optimal)
- No need for pragmas to protect MPI calls
- Constraints:
  - Ordering of MPI calls maintained within each thread but not across MPI process -- user is responsible for preventing race conditions
  - Blocking MPI calls block only the calling thread
- Multiple is rarely required; most algorithms can be written without it
Which Threading Model Should I Use?

Depends on the application!

<table>
<thead>
<tr>
<th>Model</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>Portable: every MPI implementation supports this</td>
<td>Limited flexibility</td>
</tr>
<tr>
<td>Funneled</td>
<td>Simpler to program</td>
<td>Manager thread could get overloaded</td>
</tr>
<tr>
<td>Serial</td>
<td>Freedom to communicate</td>
<td>Risk of too much cross-communication</td>
</tr>
<tr>
<td>Multiple</td>
<td>Completely thread safe</td>
<td>Limited availability; sub-optimal</td>
</tr>
</tbody>
</table>
Designing Hybrid Algorithms

- Just because you can communicate thread-to-thread, doesn’t mean you should
- Tradeoff between lumping messages together and sending individual messages
  - Lumping messages together: one big message, one overhead
  - Sending individual messages: less wait time (?)
- Programmability: performance will be great, when you finally get it working!
Example: Mesh Partitioning

- Regular mesh of finite elements
- When we partition mesh, need to communicate information about (domain) adjacent cells to (computationally) remote neighbors
Example: Mesh Partitioning
Example: Mesh Partitioning
Bibliography/Resources: OpenMP

- LLNL OpenMP Tutorial, [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/)
Bibliography/Resources: OpenMP

- OpenMP.org: https://www.openmp.org/
- OpenMP Standard: https://www.openmp.org/specifications/
  - 5.0 Specification: https://www.openmp.org/spec-html/5.0/openmp.html
  - 5.0 code examples: https://www.openmp.org/wp-content/uploads/openmp-examples-5.0.0.pdf
Bibliography/Resources: Hybrid Programming

APPENDIX 1: COMPUTING PI

“Pi” by Gregory Bastien, from
http://www.flickr.com/photos/gregory_bastien/2741729411/sizes/z/in/photostream/
Computing $\pi$

- Method of Darts is a TERRIBLE way to compute $\pi$
  - Accuracy proportional to square root of number of darts
  - For one decimal point increase in accuracy, need 100 times more darts!

- Instead,
  - Look it up on the internet, e.g., http://www.geom.uiuc.edu/~huberty/math5337/groupe/digits.html
  - Compute using BBP (Bailey-Borwein-Plouffe) formula:
    $$\pi = \sum_{n=0}^{\infty} \left( \frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left( \frac{1}{16} \right)^n$$
  - For less accurate computations, try your programming language’s constant, or quadrature or power series expansions
APPENDIX 2: ABOUT RANDOM NUMBER GENERATION

“Random Number Generator insides” by mercuryvapour, from http://www.flickr.com/photos/mercuryvapour/2743393057/sizes/l/in/photostream/
About Random Number Generation

- No such thing as random number generation – proper term is pseudorandom number generator (PRNG)
- Generate long sequence of numbers that seems “random”
- Properties of good PRNG:
  - Very long period
  - Uniformly distributed
  - Reproducible
  - Quick and easy to compute
Pseudorandom Number Generator

- Generator from \texttt{lcgenerator.h} is a Linear Congruential Generator (LCG)
  - Short period (= \texttt{PMOD}, 714025)
  - Not uniformly distributed – known to have correlations
  - Reproducible
  - Quick and easy to compute
  - Poor quality (don’t do this at home)

Correlation of RANDU LCG (source: http://upload.wikimedia.org/wikipedia/commons/3/38/Randu.png)
Good PRNGs

- For serial codes
  - Mersenne twister
  - GSL (GNU Scientific Library), many generators available (including Mersenne twister) [http://www.gnu.org/software/gsl/](http://www.gnu.org/software/gsl/)
  - Also available in Intel MKL

- For parallel codes
  - SPRNG, regarded as leading parallel pseudorandom number generator [http://sprng.cs.fsu.edu/](http://sprng.cs.fsu.edu/)