Crash Course in Supercomputing

Computing Sciences Summer Student Program & NERSC/ALCF/OLCF
Supercomputing User Training 2024

Rebecca Hartman-Baker, PhD
User Engagement Group Lead
Charles Lively III, PhD
Science Engagement Engineer
Helen He, PhD
User Engagement Group
June 28, 2024
Today’s Pipeline
Morning Session Overview

• Introduction to Parallel Programming Concepts - 09:00 am PDT
• Understanding Supercomputer Architecture
• Basic Parallelism & MPI
• BREAK - 10:30 a.m. - 10:45 a.m. PDT
• MPI Collectives
• Q&A
• LUNCH - 12:00 p.m. - 01:00 p.m. PDT

Please refer to Event Web Page for Specific Times
Today’s Pipeline

Afternoon Session Preview (after Lunch)

• Introduction to OpenMP: 01:00 p.m. PDT
• Understanding OpenMP + Hybrid OpenMP Concepts
• BREAK: 02:45 p.m. - 03:00 p.m. PDT
• Interactive Exercises & Hands-On Practice
• ADJOURN: 04:00 p.m. PDT

Please refer to Event Web Page for More Detailed Session Times
Some Logistics

- In-person attendees please also join Zoom for full participation
- Please change your name in Zoom session
  - to: first_name last_name
  - Click “Participants”, then “More” next to your name to rename
- Click the CC button to toggle captions and View Full Transcript
- Session is being recorded
- Users are muted upon joining Zoom
  - Feel free to unmute and ask questions or ask in GDoc below
- GDoc is used for Q&A (instead of Zoom chat)
  - https://tinyurl.com/4fvkzeud
- Please answer a short survey afterward
  - https://tinyurl.com/562byv62
Some Logistics

● Slides and videos will be available on NERSC Training Event page and LBNL Computing Sciences Summer Program page
  ○ https://www.nersc.gov/crash-course-in-supercomputing-jun2024/
  ○ https://cs.lbl.gov/careers/summer-student-and-faculty-program/2024-csa-summer-program/summer-program/

● You’re encouraged to register for OpenMP Monthly Training Series, May-Oct 2024
  ○ https://www.nersc.gov/openmp-training-series-may-oct-2024
  ○ Session 3 of 7 on July 8. Can catch up Session 1 and 2 via videos and exercises

● Introduction to CUDA Programming Training (coming soon)
Hands-on Exercises on Perlmutter

ssh <user>@perlmutter.nersc.gov, land on login node:

● % cd $SCRATCH
● % git clone https://github.com/NERSC/crash-course-supercosmputing.git
  ○ Downloads all exercises (and answers!)

● References
  ○ Running Jobs: https://docs.nersc.gov/jobs/
  ○ Interactive Jobs: https://docs.nersc.gov/jobs/examples/#interactive
Using Perlmutter Compute Node Reservations

- Existing NERSC users (at time of registration) have been added to “ntrain3” project
- Apply for a training account if no NERSC account at time of registration or if MFA for login is not setup yet
  - [https://iris.nersc.gov/train](https://iris.nersc.gov/train), and use the 4-letter code bk8X
  - Training accounts valid until July 10
- Perlmutter node reservations: 10:30 am - 4:30 pm PDT today
  - `--reservation=hpc_course -A ntrain3 -C cpu` for sbatch or salloc sessions
  - No need to use `--reservation` or `-A` when outside of the reservation hours
NERSC Code of Conduct

As NERSC collaborators, we are all bound by the Code of Conduct:

- Team Science
- Service
- Trust
- Innovation
- Respect

We agree to **work together professionally and productively** towards our shared goals while respecting each other’s differences and ideas.

We should all feel free to speak up to maintain this environment and remember there are resources available to **report violations** to foster an inclusive, collaborative environment. Email **nersc-training@lbl.gov** for any concerns.

[https://www.nersc.gov/nersc-code-of-conduct](https://www.nersc.gov/nersc-code-of-conduct) or search “NERSC Code of Conduct”
Introduction to Parallel Programming Concepts
I. PARALLELISM

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

- Concepts of Parallelization
- Serial vs. Parallel
- Parallelization strategies
What is Parallelism?

- Generally Speaking:
  - Parallelism lets us work smarter, not harder, by simultaneously tackling multiple tasks.
  - How?
    - the concept of dividing a task or problem into smaller subtasks that can be executed simultaneously.
  - Benefit?
    - Work can get done more efficiently, thus quicker!
Parallelization Concepts

This concept applies to both everyday activities like preparing dinner:

● Imagine preparing a lasagna dinner with multiple tasks involved.
● Some tasks, such as making the sauce, assembling the lasagna, and baking it, can be performed independently and concurrently.
● These tasks do not depend on each other's completion, allowing for parallel execution.
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 the sauce
- Assembling the lasagna
- Baking the lasagna
- Washing lettuce
- Cutting vegetables
- Assembling the salad

**PARALLEL TASKS**
- Making the lasagna
- Making the salad
- Setting the table
Serial vs. Parallel: Graph

- Make Sauce
- Cook Noodles
- Grate Cheese
- Assemble
- Bake
- Lasagna
- Wash lettuce
- Wash veg
- Cut lettuce
- Cut veg
- Assemble
- Salad
- Set Table
- Cut bread
- Spread
- Bake
- Garlic Bread
- Prep butter

Serve Dinner

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

Synchronization Points

Serve Dinner

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

Wash lettuce
- Cut lettuce
- Assemble
- Set Table

Wash veg
- Cut veg
- Assemble
- Salad

Prep butter
- Cut bread
- Spread
- Bake
- Garlic Bread
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)
Understanding Supercomputing Architecture
II. ARCHITECTURE

II. Supercomputer Architecture

- What is a supercomputer?
- Conceptual overview of architecture

Cray 1 (1976)
IBM Blue Gene (2005)
Cray XT5 (2009)

HPE-Cray Shasta Architecture (2021)
Future HPC Architecture (2029-???)
What Is a Supercomputer?

“The biggest, fastest computer right this minute.” – Henry Neeman

Tips on Identifying a Supercomputer

- Generally, at least 100 times more powerful than PC
- This field of study known as supercomputing, high-performance computing (HPC), or scientific computing
- Scientists utilize supercomputers to solve complex problems.
- Really hard problems need really LARGE (super)computers
Supercomputing Architectures

- **Cluster Architecture**
  - Connects multiple standalone computers to work together as a single system. Provides a cost-effective solution for scalable computing power.

- **Symmetric Multiprocessing (SMP)**
  - Involves multiple processors sharing a single memory space. Suitable for tasks requiring frequent communication between processors.

- **Massively Parallel Processing (MPP)**
  - Consists of many processors with their own memory. Effective for tasks that can be divided into independent subtasks.
SMP Architecture

- SMP stands for Symmetric Multiprocessing architecture
  - commonly used in supercomputers, servers, and high-performance computing environments.
  - all processors have equal access to memory and input/output devices.
    - 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.
- SMP systems and architectures allow for better load balancing and resource utilization across multiple processors.
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 very common
  - Multiple \{16, 24, 32, 64, 68, 128\}-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
NERSC Systems Ecosystem

1,792 GPU-accelerated nodes
4 NVIDIA A100 GPUs+1 AMD “Milan” CPU
448 TB (CPU) + 320 TB (GPU) memory

3,072 CPU-only nodes
2 AMD “Milan” CPUs
1,536 TB CPU memory

HPE Slingshot 11 interconnect
4 NICs/GPU node,
1 NIC/CPU node

>5 TB/s

50 GB/s

1.6 TB/s

100 GB/s

5 GB/s

2 x 400 Gb/s

2 x 100 Gb/s

Off-Platform Storage

HPSS Tape Archive ~300 PB

Common File System 130 PB

/home 450 TB

DTNs, Gateways

edge services

Science Friendly
Security
Production Monitoring
Power Efficiency

LAN

Experimental Facility
ASCR Facility
Home Institution
Cloud
Edge
First phase arrived 2021; second phase in 2022; final acceptance in 2023
• GPU-accelerated and CPU-only nodes
• HPE Cray Slingshot high-performance network
• 35 PB all-flash scratch file system

**GPU-Accelerated Nodes**
• 1,536 GPU-accelerated nodes
• 1 AMD “Milan” CPU + 4 NVIDIA A100 GPUs per node
• 256 GB CPU memory and 40 GB GPU high BW memory

**CPU-Only Nodes**
• 3,072 CPU only nodes
• 2 AMD “Milan” CPUs per node
• 512 GB memory per node
HPC Systems: Perlmutter

**GPU nodes:**
- Immense compute power from GPUs
- Large jobs using many GPUs encouraged
- Great for codes that can exploit GPU compute power

**CPU nodes:**
- Powerful CPUs (but only 10% of GPU compute power)
- Equivalent in compute power to all of Cori (former system)
- More like a traditional cluster
- Great for throughput jobs
File Systems

- Global File Systems:
  - Home
  - Community (CFS)
- Local File Systems:
  - Scratch
- Long-term Storage System:
  - HPSS
Global File Systems

Home
- Permanent, relatively small storage
- Mounted on all platforms
- NOT tuned to perform well for parallel jobs
- Quota cannot be changed
- Snapshot backups (7-day history)
- Perfect for storing data such as source code, shell scripts

Community File System (CFS)
- Permanent, larger storage
- Mounted on all platforms
- Medium performance for parallel jobs
- Quota can be changed
- Snapshot backups (7-day history)
- Perfect for sharing data within research group
Local File Systems

**Scratch**

- Large, temporary storage
- Local to machine
- Optimized for read/write operations, NOT storage
- Not backed up
- Purge policy (12 weeks)
- **Perfect for staging data and performing computations**
Long-Term Storage System

HPSS

- High-Performance Storage System
- Archival storage of infrequently accessed data
- Hierarchical storage:
  - Data first ingested onto high-performance disk arrays
  - Migrated to large enterprise tape subsystem for long-term retention
NERSC Architectures Through the Years

- **Seaborg (2003-2006):** An IBM SP system with 6,656 Power3 processors, each with 375 MHz. It used shared memory and IBM's high-performance switch (HPS) interconnect. The system delivered 10 teraflops.

- **Jacquard (2004-2007):** A Linux cluster with 712 nodes, each containing dual Intel Xeon processors (3.06 GHz). It had 4 GB of memory per node and used Myrinet interconnects, providing 9.2 teraflops.

- **Bassi (2005-2009):** An IBM Power5+ system with 888 processors (1.9 GHz). It had 8 GB of memory per processor and used IBM's Federation switch interconnect, achieving 3.6 teraflops.
NERSC Architectures Through the Years

- **Seaborg (2003-2006):** An IBM SP system with 6,656 Power3 processors, each with 375 MHz. It used shared memory and IBM's high-performance switch (HPS) interconnect. The system delivered 10 teraflops.

- **Jacquard (2004-2007):** A Linux cluster with 712 nodes, each containing dual Intel Xeon processors (3.06 GHz). It had 4 GB of memory per node and used Myrinet interconnects, providing 9.2 teraflops.

- **Bassi (2005-2009):** An IBM Power5+ system with 888 processors (1.9 GHz). It had 8 GB of memory per processor and used IBM's Federation switch interconnect, achieving 3.6 teraflops.
NERSC Architectures Through the Years

- **Seaborg (2003-2006):** An IBM SP system with 6,656 Power3 processors, each with 375 MHz. It used shared memory and IBM's high-performance switch (HPS) interconnect. The system delivered 10 teraflops.

- **Jacquard (2004-2007):** A Linux cluster with 712 nodes, each containing dual Intel Xeon processors (3.06 GHz). It had 4 GB of memory per node and used Myrinet interconnects, providing 9.2 teraflops.

- **Bassi (2005-2009):** An IBM Power5+ system with 888 processors (1.9 GHz). It had 8 GB of memory per processor and used IBM's Federation switch interconnect, achieving 3.6 teraflops.
NERSC Architectures Through the Years

- **Franklin (2008-2012):** A Cray XT4 system with 38,288 AMD Opteron cores (2.3 GHz). It used DDR2 memory and Cray's SeaStar2+ interconnect, delivering 352 teraflops.
- **Hopper (2010-2015):** A Cray XE6 system with 153,216 AMD Magny-Cours cores (2.1 GHz). It had 2 GB of memory per core and used Cray's Gemini interconnect, providing 1.28 petaflops.
NERSC Architectures Through the Years

- **Edison (2013-2019):** A Cray XC30 system with 133,824 Intel Ivy Bridge cores (2.4 GHz). It used DDR3 memory and Cray's Aries interconnect, providing 2.57 petaflops.
- **Cori (2016-2023):** A Cray XC40 system with 622,336 cores, including Intel Haswell and Knights Landing processors. It features DDR4 memory and Cray's Aries interconnect, delivering 30 petaflops.
State-of-the-Art Architectures

- Hybrid CPU/GPGPU architectures also very common
  - 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
    - Often use CUDA to directly program GPU offload portions of code
    - Alternatives: standards-based directives, OpenACC or OpenMP offloading; programming environments such as Kokkos or Raja
Introduction to Message Passing Interface (MPI)
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 Message Passing Interface (MPI)

● The Message Passing Interface (MPI) is a standardized and portable message-passing system designed to function on a wide variety of parallel computing architectures.
  ○ Standards have evolved over the years
  ○ Accommodate advances in hardware and programming practices.
● Industry standard for parallel programming
  ○ 200+ page document
Introduction to MPI

- MPI implemented by many vendors; open source implementations available too
  - Cray, IBM, HPE vendor implementations
  - MPICH, OpenMPI (open source)
- MPI function library is used in writing C, C++, or Fortran programs in HPC
Introduction to MPI

- **MPI-1 (1994 finalized and released)**
  - Provided basic point-to-point and collective communication functionalities.

- **MPI-2 (1996 release)**
  - Introduced several significant extensions, including dynamic process management, parallel I/O, and one-sided communications.

- **MPI-3 (2012 release)**
  - Further enhanced the capabilities of MPI with non-blocking collective operations, improved one-sided communications, and better support for shared memory programming. Added support for the Fortran 2008 standard.

- **MPI-4.0 (June 2021 release)**
  - Includes several enhancements and new features
MPI 4.0 Standard

- **Partitioned Communications**
  - Introduces a new communication mechanism designed for GPUs & other devices where data can be partitioned into parts that can be processed independently.

- **Persistent Collectives**
  - Extends the existing persistent communication interface to include collective operations, providing optimizations for frequently repeated operations.

- **Fault Tolerance**
  - Adds new mechanisms to handle failures in hardware and processes more effectively.

- **Enhancements for Hybrid Programming**
  - Improvements in the handling of shared memory, which is crucial for systems combining multiple levels of parallelism.
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
  - Initializes the MPI execution environment. Must be called before any other MPI function.

- **MPI_Finalize(void)** shuts down MPI
  - Place near end of code, after last MPI command
  - Terminates the MPI execution environment. No MPI function can be called after this except **MPI_Init** and **MPI_Finalize**.
Message Passing Interface

Message passing interface (MPI)

MPI_COMM_WORLD

Process ID (rank):

0
1
2
3
4
5

CLUSTER

node 1
CPU 0
CPU 1

node 2
CPU 0

node 3
CPU 0
CPU 1

node 4
CPU 0
Environmental Inquiry

- **MPI_Comm_size(MPI_Comm comm, int *size)**
  - Determines the size of the group associated with a communicator
  - Allows flexibility in number of processes used in program

- **MPI_Comm_rank(MPI_Comm comm, int *rank)**
  - Find out identifier of current process
  - Determines the rank of the calling process in the communicator.
  - $0 \leq rank \leq size-1$
Message Passing: Send

- **MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**
  - Performs a send from this MPI process to another.
  - Send message of length `count` items and datatype `datatype` contained in `buf` with tag `tag` to process number `dest` in communicator `comm`.
  - With MPI 4.0, The `buf` parameter is now marked as `const` to indicate that the buffer should not be modified during the send operation.
  - 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)
  - Performs a blocking receive of data from another process.
  - 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!** Standard receive function is blocking
- **MPI_Recv** returns only after receive buffer contains requested message
- **MPI_Send** *may or may not block* until message received (usually blocks)
  - Depends on implementation standard as the blocking behavior of **MPI_Send** depends on the size of the message and the underlying system's buffering capabilities.
  - **MPI_Send** will block until it can safely copy the message to the system's buffer, which might not necessarily mean the message has been received by the destination process.
  - For small messages, it may return quickly if the system can buffer them, but for larger messages, it may block until the receiving process calls **MPI_Recv**.
- **Must watch out for deadlock**
Warning: DEADLOCKS

Must Watch Out for DEADLOCKS

- Deadlocks can occur in MPI programs if send and receive operations are not properly ordered
  - more generally, if processes are waiting on each other indefinitely.
- To avoid deadlocks, ensure that the send/receive operations are properly matched
  - And consider using non-blocking communication functions (MPI_Isend, MPI_Irecv) or changing the program's structure to avoid circular dependencies.
Deadlocking Example (Always)

```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;
}
```
```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;
}
Explanation: 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 $\frac{\pi R^2}{4R^2} = \frac{\pi}{4}$

"Dartboard" by AndyRobertsPhotos, from http://www.flickr.com/photos/aroberts/2907670014/sizes/o/in/photostream/
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 and Charles, 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 "lcg_generator.h"
static long num_trials = 1000000;

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

    for (long 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;
}
Serial Code (lcgenerator.h)

// 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
Partition

“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
“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 π
- Every processor, including manager, will perform equal share of dart throws
Your Assignment

- Clone the whole assignment (including answers!) to Perlmutter from the repository with: `git clone https://github.com/NERSC/crash-course-supercomputing.git`
- Copy `darts.c/lcgenerator.h` or `darts.f` (your choice) from `crash-course-supercomputing/darts-suite/{c,for tran}`
- Parallelize the code using the 6 basic MPI commands
- Rename your new MPI code `darts-mpi.c` or `darts-mpi.f90`
Introduction to MPI Collectives
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 $N_P$ equal pieces, with $i$th segment sent to $i$th process in group
- `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: `MPI_Scatterv(...)`
- `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** (float and int)
    - **MPI_DOUBLE_INT** (double and int)
    - **MPI_LONG_INT** (long and int)
    - **MPI_2INT** (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 \( \text{recvbuf} \)
- `int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
- Corresponding `MPI_Alltoallv` function also available
All-Reduce

- Same as `MPI_Reduce` except result appears on all processes
- `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://hpc-tutorials.llnl.gov/mpi/](https://hpc-tutorials.llnl.gov/mpi/)

- MPI Standard at MPI Forum: [https://www.mpi-forum.org/docs/](https://www.mpi-forum.org/docs/)
  - MPI 3.1: [https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf](https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf)
  - MPI 4.0: [https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf](https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf)
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.f90`