Outline

- Quick review of hardware architectures
- Running on supercomputers
- Message Passing
- MPI
Parallel Architecture Types

- Uniprocessor
  - Scalar processor
  - Vector processor
  - Single Instruction Multiple Data (SIMD)

- Shared Memory Multiprocessor (SMP)
  - Shared memory address space
  - Bus-based memory system
  - Interconnection network
Parallel Architecture Types (2)

- Distributed Memory Multiprocessor
  - Message passing between nodes

- Cluster of SMPs
  - Shared memory addressing within SMP node
  - Message passing between SMP nodes

- Massively Parallel Processor (MPP)
  - Many, many processors

- Can also be regarded as MPP if processor number is large
Parallel Architecture Types (3)

- Multicore
  - Multicore processor
    - Cores can be hardware multithreaded (hyperthread)
  - GPU accelerator
  - “Fused” processor accelerator

- Multicore SMP+GPU Cluster
  - Shared memory addressing within SMP node
  - Message passing between SMP nodes
  - GPU accelerators attached
How do you get parallelism in the hardware?

- Instruction-Level Parallelism (ILP)
- Data parallelism
  - Increase amount of data to be operated on at same time
- Processor parallelism
  - Increase number of processors
- Memory system parallelism
  - Increase number of memory units
  - Increase bandwidth to memory
- Communication parallelism
  - Increase amount of interconnection between elements
  - Increase communication bandwidth
Parallelism on Distributed Memory Machines

- Each processing element is constrained by what data of the application it can see natively
  - That is, what it can get to with referencing memory
- Machine scale has gone up considerably
- Penalty for coordinating with other processing elements is now significantly higher
  - Approaches change accordingly
Why Large-scale Applications?

- Why simulation?
  - Simulations are sometimes more cost effective than experiments
  - Sometimes experiments cannot be performed at all

- Why extreme scale?
  - More compute cycles, more memory, and so on …
  - Necessary for faster and/or more accurate simulations
Large-scale Supercomputers

- How big are these machines?
  - Measured in “FLOPs”
    - floating point operations per second
  - 1 GigaFLOP = 1 billion FLOPs
  - 1 TeraFLOP = 1000 GigaFLOPs
  - 1 PetaFLOP = 1,000,000 GigaFLOPs
    - where we are today
  - 1 ExaFLOP = 1,000,000,000 GigaFLOPs
    - potentially arriving as soon as 2018
**Distributed Memory Multiprocessors**

- Each processor has a local memory
  - Physically separated memory address space
- Processors must communicate to access non-local data
  - Message communication (message passing)
    - *Message passing architecture*
  - Processor interconnection network
- Parallel applications must be partitioned across
  - Processors: execution units
  - Memory: data partitioning
- Scalable architecture
  - Small incremental cost to add hardware (cost of node)
**Distributed Memory (MP) Architecture**

- Nodes are complete computer systems
  - Including I/O
- Nodes communicate via interconnection network
  - Standard networks
  - Specialized networks
- Network interfaces
  - Communication integration
- Easier to build
Performance Metrics: Latency and Bandwidth

- Bandwidth
  - Need high bandwidth in communication
  - Match limits in network, memory, and processor
  - Network interface speed vs. network bisection bandwidth

- Latency
  - Performance affected since processor may have to wait
  - Harder to overlap communication and computation
  - Overhead to communicate is a problem in many machines

- Latency hiding
  - Increases programming system burden
  - Examples: communication/computation overlaps, prefetch
Advantages of Distributed Memory Architectures

- The hardware can be simpler (especially versus NUMA) and is more scalable
- Communication is explicit and simpler to understand
- Explicit communication focuses attention on costly aspect of parallel computation
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization
- Easier to use sender-initiated communication, which may have some advantages in performance
Outline

- Quick review of hardware architectures
- Running on supercomputers
  - The purpose of these slides is to give context, not to teach you how to run on supercomputers
- Message Passing
- MPI
Running on Supercomputers

- Sometimes one job runs on the entire machine, using all processors
  - These are called “hero runs”…

- Sometimes many smaller jobs are running on the machine

- For most supercomputer, the processors are being used nearly continuously
  - Processors are the “scarce resource”
  - Jobs to run on them are “plentiful”
Running on Supercomputers

- You plan a “job” you want to run
  - The job consists of a parallel binary program and an “input deck” (something that specifies input data for the program)

- You submit that job to a “queue”

- The job waits in the queue until it is scheduled

- The scheduler allocates resources when (i) resources are available and (ii) the job is deemed “high priority”
Running on Supercomputers

- The scheduler runs scripts that initialize the environment
  - Typically done with environment variables
- At the end of initialization, it is possible to infer:
  - What the desired job configuration is (i.e., how many tasks per node)
  - What other nodes are involved
  - How your node’s tasks relates to the overall program
- The MPI library knows how to interpret all of this information and hides the details from you
Job submission on ACISS

Torque/PBS Queueing System

Submitting jobs

ACISS uses a queuing system to schedule and control use of system resources. Like nearly all modern supercomputers, it is a cluster composed of nodes, and nodes in turn have other properties (such as number of cores or GPUs).

Programs to be run on the nodes are quantized in the form of jobs. PBS (Portable Batch System) is the frontend used to requisition resources for jobs. The user frontend is the qsub command, which submits a script along with a list of requested resources. This then goes into the queue and is executed at the scheduler’s leisure. The second key user command is qstat, which prints the status of jobs and queues.

qsub has a lot of options (which the man page and the Interwebs will happily explain in great detail) but the most important are:

- "-q xyz": Requests the job be put in the queue ‘xyz’. A list of queues is available from ‘qstat -q’
- "-l": Requesting an interactive session.
- "-lX": Indicates graphics should be forwarded such that GUI programs will work from the interactive session. This requires that you ssh’ed into Aciss with -X or -Y as well in order to work.
- "-l resource=amount,resource=amount...": Requests resources such as number of nodes (nodes=X), number of cores per node (ppn=Y), amount of memory (mem=Z), or a feature such as “scratch” or “mpi”. The most common form is nodes=X;ppn=Y, to reserve X nodes and Y cores on each node. Users are given 1 node and 12 cores by default.

qsub options can be embedded in the submission script via PBS directives, or fed into the qsub command line. Thus you could do

```bash
qsub ./myscript.sh
    programX | qsub -q generic -l nodes=1:ppn=12,mem=20gb
    qsub -q generic -l nodes=1:ppn=12 < evilplan.txt
```

Once your job has been submitted, you will be given a jobid associated with that job. You can check the status of your job using ‘qstat <jobid>’.
Job submission on ACISS

Queues

The available queues can be enumerated in detail by the ‘qstat -q’ command. These are the queues available on ACISS:

- generic: nodes with 12 cores and 72GB ram
- fatnodes: nodes with 32 cores and 384GB of ram
- gpu: nodes with 12 cores, 72GB ram, and 3 nVidia M2070 GPUs (with 512 stream processors and 6GB GDDR5 each)

These queues limit jobs to one day of wallclock time. The default queue is **generic**.

Here are some additional queues:

- longgen/longfat/longgpu: generic/fat/gpu nodes with 4 day time limit
- xlonggen/xlongfat/xlonggpu: generic/fat/gpu nodes with 2 week time limit
- short: generic node with 4 hour time limit
- student: node type may vary, reserved for students only (1 day)

Some info on the different queues is nicely tabulated by `qstat -q`.

**Please avoid using the long or xlong queues unless your job needs them.**

**Only 1-day and short queues can be booked in interactive (-l) mode.**
Outline

- Quick review of hardware architectures
- Running on supercomputers
- Message Passing
- MPI
Acknowledgements and Resources

- Portions of the lectures slides were adopted from:
  - Argonne National Laboratory, MPI tutorials.
  - Lawrence Livermore National Laboratory, MPI tutorials
  - See online tutorial links in course webpage


Types of Parallel Computing Models

- **Data parallel**
  - Simultaneous execution on multiple data items
  - Example: Single Instruction, Multiple Data (SIMD)

- **Task parallel**
  - Different instructions on different data (MIMD)

- **SPMD (Single Program, Multiple Data)**
  - Combination of data parallel and task parallel
  - Not synchronized at individual operation level

- **Message passing** is for MIMD/SPMD parallelism
  - Can be used for data parallel programming
The Message-Passing Model

- A process is a program counter and address space
- Processes can have multiple threads (program counters and associated stacks) sharing a single address space

- MPI is for communication among processes (ranks)
  - More recent MPI versions allow threads to call MPI routines
- Interprocess communication consists of
  - Synchronization
  - Data movement
SPMD

- Data distributed across processes
  - Not shared

“Owner compute” rule: Process that “owns” the data (local data) performs computations on that data.
Message Passing Programming

- Defined by communication requirements
  - Data communication (necessary for algorithm)
  - Control communication (necessary for dependencies)
- Program behavior determined by communication patterns
- Message passing infrastructure attempts to support the forms of communication most often used or desired
  - Basic forms provide functional access
    - can be used most often
  - Complex forms provide higher-level abstractions
    - serve as basis for extension
    - example: graph libraries, meshing libraries, ...
  - Extensions for greater programming power
Communication Types

- Two ideas for communication
  - Cooperative operations
  - One-sided operations
Cooperative Operations for Communication

- Data is cooperatively exchanged in message-passing
- Explicitly *sent* by one process and *received* by another
- Advantage of local control of memory
  - Any change in the receiving process’s memory is made with the receiver’s explicit participation
- Communication and synchronization are combined

![Diagram showing communication between Process 0 and Process 1](image-url)
One-Sided Operations for Communication

- One-sided operations between processes
  - Include remote memory reads and writes
- Only one process needs to explicitly participate
  - There is still agreement implicit in the SPMD program
- Advantages?
  - Communication and synchronization are decoupled

```
Process 0
Put(data)
(memory)

Process 1
(memory)
Get(data)
```

time
Pairwise vs. Collective Communication

- Communication between process pairs
  - Send/Receive or Put/Get
  - Synchronous or asynchronous (we’ll talk about this later)

- Collective communication between multiple processes
  - Process group (collective)
    - several processes logically grouped together
  - Communication within group
  - Collective operations
    - communication patterns
      - broadcast, multicast, subset, scatter/gather, …
    - reduction operations
Outline

- Quick review of hardware architectures
- Running on supercomputers
- Message Passing
- MPI
What is MPI (Message Passing Interface)?

- Message-passing library (interface) specification
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- Targeted for parallel computers, clusters, and NOWs
  - NOWs = network of workstations
- Specified in C, C++, Fortran 77, F90
- Full-featured and robust
- Designed to access advanced parallel hardware
- End users, library writers, tool developers
- Message Passing Interface (MPI) Forum
  - [http://www.mpiforum.org/docs/docs.html](http://www.mpiforum.org/docs/docs.html)
Why Use MPI?

- Message passing is a mature parallel programming model
  - Well understood
  - Efficient match to hardware (interconnection networks)
  - Many applications
- MPI provides a powerful, efficient, and portable way to express parallel programs
- MPI was explicitly designed to enable libraries …
- … which may eliminate the need for many users to learn (much of) MPI
- Need standard, rich, and robust implementation
- Three versions: MPI-1, MPI-2, MPI-3 (new features!)
  - Robust implementations including free MPICH (ANL)
Features of MPI

- General
  - *Communicators* combine context and group for security
  - Thread safety (implementation dependent)

- Point-to-point communication
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal, synchronous, ready, buffered

- Collective
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology
Features of MPI (continued)

- Application-oriented process topologies
  - Built-in support for grids and graphs (based on groups)

- Profiling
  - Hooks allow users to intercept MPI calls
  - Interposition library interface (PMPI)
  - Many tools (e.g., TAU) use PMPI

- Environmental
  - Inquiry
  - Error control
Is MPI Large or Small?

- MPI is large
  - MPI-1 is 128 functions, MPI-2 is 152 functions
  - Extensive functionality requires many functions
  - Not necessarily a measure of complexity
- MPI is small (6 functions)
  - Many parallel programs use just 6 basic functions
- “MPI is just right,” said Baby Bear
  - One can access flexibility when it is required
  - One need not master all parts of MPI to use it
To Use or Not Use MPI? That is the Question?

- **USE**
  - You need a portable parallel program
  - You are writing a parallel library
  - You have irregular or dynamic data relationships that do not fit a data parallel model
  - You care about performance and have to do Exercise 1

- **NOT USE**
  - You don’t need parallelism at all (Ha!)
  - You can use libraries (which may be written in MPI)
  - You can use multi-threading in a concurrent environment
Getting Started

- Writing MPI programs
- Compiling and linking
- Running MPI programs
A Simple MPI Program (C)

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

What does this program do?
A Simple MPI Program (C++)

```cpp
#include <iostream.h>
#include "mpi++.h"

int main( int argc, char *argv[] )
{
    MPI::Init(argc,argv);
    cout << "Hello, world!" << endl;
    MPI::Finalize();
    return 0;
}
```
A Minimal MPI Program (Fortran)

```fortran
program main
use MPI
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```
MPI_Init

- What happens during MPI initialization?
- Think about it
- How do hardware resources get allocated?
  - Hmm, is this part of MPI?
- How do processes on different nodes get started?
  - Where does their executable program come from?
- What do the processes need to know?
- What about OS resources?
- What about tools that are running with MPI?
- …
MPI_Finalize

- Why do we need to finalize MPI?
- What happens during MPI finalization?
- Think about it
- What is necessary for a “graceful” MPI exit?
  - Can bad things happen otherwise?
  - Suppose the one process exits?
- How do resources get de-allocated?
- What about communications?
- What type of exit protocol might be used?
- What about tools?
Notes on C and Fortran

- C and Fortran library bindings correspond closely

- In C:
  - `mpi.h` must be included
  - MPI functions return error codes or `MPI_SUCCESS`

- In Fortran:
  - `mpif.h` must be included, or use MPI module (MPI-2)
  - All MPI calls are to subroutines
    - place for the return code in the last argument

- C++ bindings, and Fortran-90 issues, are part of MPI-2
Error Handling

- By default, an error causes all processes to abort
- The user can cause routines to return (with an error code)
  - In C++, exceptions are thrown (MPI-2)
- A user can also write and install custom error handlers
- Libraries may handle errors differently from applications
Running MPI Programs

- MPI-1 does not specify how to run an MPI program
- Starting an MPI program is dependent on implementation
  - Scripts, program arguments, and/or environment variables
- `% mpirun -np <procs> a.out`
  - For MPICH under Linux
- `mpiexec <args>`
  - Recommended part of MPI-2, as a recommendation
  - `mpiexec` for MPICH (distribution from ANL)
  - `mpirun` for SGI’s MPI
Finding Out About the Environment

- Two important questions that arise in message passing
  - How many processes are being used in computation?
  - Which one am I?

- MPI provides functions to answer these questions
  - `MPI_Comm_size` reports the number of processes
  - `MPI_Comm_rank` reports the rank
    - number between 0 and size-1
    - identifies the calling process
Better “Hello World” (C)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

☐ What does this program do and why is it better?
MPI Basic Send/Receive

- We need to fill in the details in:

  Process 0
  \[\text{Send}(\text{data})\]

  Process 1
  \[\text{Receive}(\text{data})\]

- Things that need specifying:
  - How will “data” be described?
  - How will “processes” be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?
What is message passing?

- Data transfer plus synchronization

- Requires cooperation of sender and receiver
- Cooperation not always apparent in code
Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context
  - Must be received in the same context!
- A group and context together form a communicator
- A process is identified by its rank
  - With respect to the group associated with a communicator
- There is a default communicator
  - `MPI_COMM_WORLD`
  - Contains all initial processes
**MPI Datatypes**

- Message data (sent or received) is described by a triple
  - address, count, datatype

- An MPI *datatype* is recursively defined as:
  - Predefined data type from the language
  - A contiguous array of MPI datatypes
  - A strided block of datatypes
  - An indexed array of blocks of datatypes
  - An arbitrary structure of datatypes

- There are MPI functions to construct custom datatypes
  - Array of (int, float) pairs
  - Row of a matrix stored columnwise
**MPI Tags**

- Messages are sent with an accompanying user-defined integer *tag*  
  - Assist the receiving process in identifying the message

- Messages can be screened at the receiving end by specifying a specific tag  
  - `MPI_ANY_TAG` matches any tag in a receive

- Tags are sometimes called “message types”  
  - MPI calls them “tags” to avoid confusion with datatypes
MPI Basic (Blocking) Send

MPI_SEND (start, count, datatype, dest, tag, comm)

- The message buffer is described by:
  - `start`, `count`, `datatype`
- The target process is specified by `dest`
  - Rank of the target process in the communicator specified by `comm`
- Process blocks until:
  - Data has been delivered to the system
  - Buffer can then be reused
- Message may not have been received by target process!
MPI Basic (Blocking) Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Process blocks (waits) until:
  - A matching message is received from system
    - Matches on source and tag
  - Buffer must be available
- source is rank in communicator specified by comm
  - Or MPI_ANY_SOURCE
- Status contains further information
- Receiving fewer than count is OK, more is not
Retrieving Further Information

- Status is a data structure allocated in the user’s program
- In C:

```c
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status );
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```
Why Datatypes?

- All data is labeled by type in MPI
- Enables heterogeneous communication
  - Support communication between processes on machines with different memory representations and lengths of elementary datatypes
  - MPI provides the representation translation if necessary
- Allows application-oriented layout of data in memory
  - Reduces memory-to-memory copies in implementation
  - Allows use of special hardware (scatter/gather)
Tags and Contexts

- Separation of messages by use of tags
  - Requires libraries to be aware of tags of other libraries
  - This can be defeated by use of “wild card” tags

- Contexts are different from tags
  - No wild cards allowed
  - Allocated dynamically by the system
  - When a library sets up a communicator for its own use

- User-defined tags still provided in MPI
  - For user convenience in organizing application

- Use `MPI_Comm_split` to create new communicators
Programming MPI with Only Six Functions

- Many parallel programs can be written using:
  - `MPI_INIT()`
  - `MPI_FINALIZE()`
  - `MPI_COMM_SIZE()`
  - `MPI_COMM_RANK()`
  - `MPI_SEND()`
  - `MPI_RECV()`

- What might be not so great with this?
  - Point-to-point (send/recv) isn’t the only way...
    - Add more support for communication
Introduction to Collective Operations in MPI

- Called by all processes in a communicator

- **MPI_BCAST**
  - Distributes data from one process (the root) to all others

- **MPI_REDUCE**
  - Combines data from all processes in communicator
  - Returns it to one process

- In many numerical algorithms, **SEND/RECEIVE**
  can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency
Summary

- The parallel computing community has cooperated on the development of a standard for message-passing libraries
- There are many implementations, on nearly all platforms
- MPI subsets are easy to learn and use
- Lots of MPI material is available