Compsci 590.3: Introduction to Parallel Computing

Alvin R. Lebeck

Slides from the University of Oregon
Admin

• Logistics
• Homework #4
• Projects
• Project work days

• Outline
• Parallel Hardware and scalability
• Message Passing
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

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

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

  – 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/MIC 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
Distributed Memory

- Each processing elements cannot access all data natively
- The scale can go up considerably
- Penalty for coordinating with other processing elements is now significantly higher
  - Approaches change accordingly
- Distributed shared memory
  - Still one physical address space (use ld/st) to communicate
- Distributed memory (shared nothing)
  - Each node has one physical address space
- Communication, sharing, and synchronization with loads/stores (reads/writes) on shared variables
Message Passing

- Communication, sharing, and synchronization with explicit messages
Programming Model vs. Architecture

- Shared memory and message passing are programming models
  - How the programmer communicates, moves data, synchronizes, coordinates, etc.

- Distributed shared memory vs. distributed memory (shared nothing) are hardware architectures

- Can implement either programming model on either architecture!
Scientific Simulation and Supercomputers

- Why simulation?
  - Simulations are sometimes more cost effective than experiments

- Why extreme scale?
  - More compute cycles, more memory, etc., lead to faster and/or more accurate simulations
How big are supercomputers?

- Measured in “FLOPs”
  - Floating point Operations Per second
    - 1 GigaFLOP = 1 billion FLOPs
    - 1 TeraFLOP = 1000 GigaFLOPs
    - 1 PetaFLOP = 1000 TeraFLOPs
      - where we are today
    - 1 ExaFLOP = 1000 PetaFLOPs
      - potentially arriving in 2018
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
Distributed Memory (shared nothing) 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)
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 (shared nothing) Architectures

- The hardware can be simpler 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
- 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


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)

- **Can use Shared Memory or Message Passing for both types of parallelism**

- **SPMD (Single Program, Multiple Data)**
  - Combination of data parallel and task parallel
  - Not synchronized at individual operation level
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
  - Not threads
- Interprocess communication consists of
  - Synchronization
  - Data movement
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 with Send and Receive functions]
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

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Put(data)</strong></td>
<td>(memory)</td>
</tr>
<tr>
<td>(memory)</td>
<td>Get(data)</td>
</tr>
<tr>
<td>time</td>
<td>time</td>
</tr>
</tbody>
</table>
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
- 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.mpi-forum.org/docs/docs.html](http://www.mpi-forum.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 (just released!)
  - 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
#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?
#include <iostream.h>
#include "mpi++.h"

int main( int argc, char *argv[] )
{
    MPI::Init(argc,argv);
    cout << "Hello, world!" << endl;
    MPI::Finalize();
    return 0;
}
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?
Error Handling

- MPI functions return error codes or MPI_SUCCESS
- 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
  - mpirun for MPICH (distribution from ANL)
  - mpirun for SGI’s MPI
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?
We need to fill in the details in:

- Process 0
  - Send(data)

- Process 1
  - Receive(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 \texttt{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
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.