Introduction to Parallel Programming

Introduction to Concepts and Methods

Argonne National Laboratory
Intent

- Assuming some knowledge about serial computing, walk away from these lectures with a basic understanding of:
  - Parallel programming terminology
  - Parallel computing architectures
  - Parallel programming models
  - Some existing methods and tools to help
The Laboratory Computing Resource Center (LCRC)

- Founded in 2002 to promote wide spread use of high performance computing across the lab
- LCRC Application Engineers
- Jazz
  - **Compute** - 350 nodes, each with a 2.4 GHz Pentium Xeon
  - **Memory** - 175 nodes with 2 GB of RAM, 175 nodes with 1 GB of RAM
  - **Storage** - 20 TB of clusterwide disk: 10 TB GFS and 10 TB PVFS
  - **Network** - Myrinet 2000
Outline

- Define Parallel Programming
- Parallel Architecture
- Parallel Programming Models
- When to Parallelize
- Overview: How to Parallelize Code
What is Parallelism?
Serial Computing

- Traditional computing model
  - One CPU
  - One copy of the data
- The CPU performs one operation at a time.
  - Single Instruction, Single Data (SISD)
- Single task is implemented
Parallel Computing

- Simultaneous use of multiple computing resources to solve a problem.

- Parallel computing
  - Breaks serial tasks into multiple tasks
  - Works on tasks simultaneously
  - Coordinates those tasks
Why Parallel Compute?

- Potential faster time to solution
  - CPU limitations
- Solve larger problems
  - Memory limitations
- Cost savings
  - Cheap PCs linked rather than more expensive architectures
Parallel Computing Hardware

- Single computer, multiple CPUs
- Multiple computers, connected by network
- A combination of the two
Parallel Programming Model

- Defines how the programmer creates and coordinates parallel tasks
- Architecture, libraries, compilers, tools
  - Create the model
  - Directs parts of algorithm
  - Does not map one to one to architecture
Basic Parallel Architecture
Why Talk About Architecture?

- Helps understand methods of parallelizing codes
- Useful when parallelizing algorithms
  - Can make parallel choices easier
- Useful for performance
- Get the buzz words
Architecture Taxonomy

- **SISD**: Single Instruction, Single Data
  - Traditional Serial Computing

- **SIMD**: Single Instruction, Multiple Data
  - Vector Pipelines

- **MISD**: Multiple Instruction, Single Data
  - Rare

- **MIMD**: Multiple Instruction, Multiple Data
  - Standard Parallel Computers
SISD

- Standard serial computing model
- One copy of the data
- Sequentially compute
  - One instruction per clock cycle
SIMD

- Like SISD, but, each instruction can operate on multiple data
- Multiple data streams, one instruction
- Vector machine
MIMD

- Very common: All multiple processor configurations
- Each processor executes a totally independent instruction stream on independent data stream
- Can be affordable
- Requires load balancing
- Can be difficult to program
MIMD : Shared Memory

- Symmetric Multiprocessor
  - SMP
- Each CPU is operating on different data, even though it is all in the same physical memory
- Pros
  - Easy for user to utilize
  - Data sharing is fast
- Cons
  - Bandwidth clogs
MIMD : Distributed Memory

- Massively Parallel Processors (MPP)
- Multiple SISD systems linked together with a network.
- The network moves the model from SISD to a MIMD
- Pro: Memory scalable
- Con
  - Harder to program
  - Bandwidth
MIMD : Hybrid Distributed Memory

- MPP Memory Arch.
- Multiple SMP-like systems (PCs) linked together with a network.
- The network moves the model from SISD to a MIMD
- Hybrid of shared memory and distributed
Memory Architectures

- **Shared Memory**
  - Multiple processors, one memory
    - Global address space
  - Each process has equal access to memory
  - UMA, CC-UMA, NUMA, CC-NUMA

- **Distributed Memory**
  - Each processor has own address space
  - Requires communication/messages to exchange data with other processors
Parallel Programming Models
What is a Parallel Programming Model?

- Method by which the programmer creates parallel tasks
  - An abstraction of the architecture
- Models do not map one-to-one to architectures
- How to choose the model?
  - Architecture
  - Algorithm
  - Capabilities/Time/Resources
High-Level Models

- **SPMD : Single Process, Multiple Data**
  - Every process
    - running the same executable
    - has different data
    - can have logic to allow different processes to perform different tasks
  - MPI

- **MPMD : Multiple Process, Multiple Data**
  - Every process
    - can execute the same or a different executable
    - will work on different data
High-Level Models - Schematic

SPMD
a.out a.out a.out ...

MPMD
a.out b.out c.out ...

tasks
Lower Level Models

- **Shared Memory**
  - Global memory space

- **Data Parallel**
  - Shared or Distributed memory splitting a larger data structure

- **Message Passing**
  - MPI, PVM, libraries using message passing

- **Threads**
  - One process, one memory space, multiple threads

- **Hybrid**
  - Ex: Threads and Message Passing
Shared Memory and Threads

- **Shared Memory** - `shmem`
  - Can be very easy to use
  - Hardware often more expensive

- **Threads** - OpenMP, POSIX
  - Used on shared memory and hybrid systems
  - Harder to use, but, very flexible
  - Threads + Message Passing
    - One of the best ways to make use of hybrid systems
Data Parallel

- Each processor works on different part of the same data structure
  - Data split across processors
  - Messages are invisible
    - Often built on top of message passing library
- SPMD approach
Data Parallel Implementations

- Data parallel constructs added to code and compiled with data parallel compiler

- Implementations
  - F90/HPF Implementations
  - Global Arrays
Message Passing

- Set of processors only using local memory
- Processors communicate
  - Send/Receive data
  - Synchronization
- Library Implementations
  - MPI, PVM
Pro’s and Cons

Shared Memory and Data Parallel
- **Pro: Easy for programmer**
  - Normally done mostly by compiler
- **Con: Memory ownership less clear**
- **Con: Potential higher cost of hardware**

Threads and Message Passing
- **Pro: Can make excellent use of hybrid architectures**
- **Con: Everything done explicitly by programmer**
  - Need a good deal of knowledge to implement well
Automatic Parallelization

- **Fully Automatic**
  - Compiler parses code
- **Programmer Directed**
  - In-line directives
  - Compiler flags
Problems with Automatic

- Wrong results may be produced
- Performance may degrade
- Limited to what it can identify
  - Loops
  - Complex code not helped
- Most automatic parallelization tools are for Fortran (HPF)
  
  So, we go with manual methods
How To Parallelize
When to Parallelize - Revisited

- Limited by memory
  - After fully optimized serial version
    - CPU and Memory usage
- Potential faster time to solution
  - Dependant on algorithm.
    - Without care, time could be longer!
      - Problem Size, Algorithm, Hardware capabilities
- Resources
Drawbacks to Parallelizing

- **Time**
  - Learning, Implementing, Debugging
- **Program Complexity**
  - Algorithms/Flow can be less clear
  - Need more software support
- **Reduces portability / reusability**
  - Tied to software availability
  - Performance for few architectures
Parallelization Process

- What code needs to be parallel and why
  - PDE solve, search, image analysis, FFT …
- Has someone else done the work?
  - Google is your friend
- Write the code
  - If existing in serial, fully debug and optimize
- Debug, Test, and Optimize
Design Considerations

- **Granularity**
  - Ratio between computation and communication
  - Fine Grain/Coarse Grain

- **Data dependency**

- **I/O**

- **Deadlock**
  - Parallelism gets trapped
New Code Components

- Initialization
- Query parallel state
  - Identify process
  - Identify number of processes
- Exchange data between processes
  - Local, Global
- Synchronization
  - Barriers, Blocking Communication, Locks
- Finalization
The Basic MPI Code

program hello
implicit none
include 'mpif.h'
integer id,nprocs,ierr
call MPI INIT(ierr)
call MPI COMM SIZE(MPI COMM WORLD,nprocs,ierr)
call MPI COMM RANK(MPI COMM WORLD,id,ierr)
print*,""Hello from processor",id,"out of",nprocs
call MPI FINALIZE(ierr)
end
I/O

- I/O is design and performance limiter
  - Large communication and time costs
- Libraries in serial and parallel
  - NetCDF, HDF5
  - Even in serial can help with design
- Parallel File Systems improving
- Parallelizing I/O a big topic
  - Similar process to parallelizing code
  - Keep I/O to a minimum
Deadlock

- A condition where two or more tasks are waiting for a message that will never come

<table>
<thead>
<tr>
<th>Task 1</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 4</td>
<td>Y = 4</td>
</tr>
</tbody>
</table>

- Change order of send/recevs
- Change algorithm
- Change to non-blocking communication
Performance Considerations

- Amdahl’s Law
  - Potential code speedup is a function of the code that can be parallelized (P)
  - $\text{speedup} = \frac{1}{\frac{P}{N} + S}$
  - N: Number of processes
  - S: 1-P, fraction of code that is serial

- Load Balancing
- Communication/Bandwidth
- I/O
Load Balancing

- Might be the hardest component
  - Balance communication/computation
- Dependant
  - Requirements of algorithm
  - Hardware
    - Memory/CPU changes
- Performance studies are your friend
Steps to Parallelize

- Identify computational hotspots in code
- Partition problem into smaller tasks
- Identify communication between tasks
- Agglomerate tasks into even larger tasks
- Map tasks to processors
Steps to Parallelize

Partition

Problem

Communicate

Agglomerate

Map

Jazz LCRC
To Start

- Understand the algorithm
  - Cannot parallelize a code without understanding how it works.
  - Might need entirely new algorithm

- Programming Model
  - We will go with MP examples
Parallel Pi
Simple First Example

“Monte Carlo” Pi
- Enscribe a circle in a square
- Randomly generate points in the square
- Determine points also in the circle
- \[ R = \frac{\text{Points in circle}}{\text{Points in square}} \]
- \( \pi \approx 4R \)

- Very easily parallel
Monte Carlo Pi

- $\pi \approx 4 \left( \frac{\text{Points in Circle}}{\text{Points in Square}} \right)$
Pi: Serial Pseudo Code

NumPoints = 10000
NumInCircle = 0
do count = 1, NumPoints
   generate random number from 0-1
   xcoordinate = random1
   generate random number from 0-1
   ycoordinate = random2
   if coords are inside circle then
      NumInCircle++
   end if
end do
PI = 4.0*NumInCircle/NumPoints
Parallel Pi Pseudocode

NumPoints = 10000; NumInCircle = 0
NumProcs = number of processors
localNumPoints = NumPoints/NumProcs
find out if I am MASTER or WORKER

do j = 1,localNumPoints
   generate random coordinates
   if random coords are inside circle
      NumInCircle++
   end if
end do

if I am MASTER
   Receive NumInCircle from all processors
   compute PI
else if I am WORKER
   Send NumInCircle to master
endif
Understand the Algorithm

- The Method
  - Computation and data components
  - New methods
- The Performance
  - Where time is spent (Hotspots)
  - gprof

The Loop:
```
do count=1,numPoints
  random coords
  if coords in circle then
    These
  else
    Not these
  end Loop
```
Partition the Algorithm

- Functional partition
  - Tasks based on function
  - Can be totally different functions
  - Ex: subroutines

- Data partition
  - Splitting an array

- Both

Functional:
Loops over only localNumPoints
Calculate pi from points
Domain Decomposition

- A common data partitioning
- Huge impact of load balancing
  - Work per unit cell
  - Dynamic assignment/grid

- Problem Data Set
- Task 1
- Task 2
- Task 3
- Task 4

1D
- BLOCK

2D
- BLOCK, C
- C, BLOCK
- BLOCK, BLOCK
- CYCLIC, C
- C, CYCLIC
- CYCLIC, CYCLIC
Functional Decomposition

- Especially useful when there is less data dependency

Diagram:
- Problem Instruction Set
- Task 1
- Task 2
- Task 3
- Task 4
Communication Patterns

- **Point to Point (Bcast)**

- **One to All (Broadcast)**

- **One to All Personalized (Scatter)**

- **Collective (Gather)**

- **All to All**

- **Shift**
Communicators

Groups of processors

COMM1

MPI_COMM_WORLD

COMM2
Communication

- Same patterns for MP and data parallel
- Local Communication
  - Tasks communicate with small numbers of local tasks
- Global Communication
  - One or more tasks talk all non-local tasks
- Communication patterns
  - Static or runtime?

Each processor sends localNumPoints to master
(can be collective pattern)
Two Common Global Ops

MPI BCAST(buffer, count, data type, root, communicator)

- MPI REDUCE(send buffer, recv buffer, count, data type, operation, root, communicator)
  - MPI SUM
  - MPI PROD
  - MPI MIN
  - MPI MAX
Communication Considerations

- Cost of communication
- Latency vs. Bandwidth
- Visibility of communication
  - Can you stop worrying here?
- Blocking vs. Non-Blocking
  - Synchronous vs. Non-Synchronous
- Scale of communication
  - Local vs. global
- Efficiency
  - Hardware, software, communication method
Group Tasks

- Agglomerate with algorithm and performance in mind
- What is worth replicating?
  - Sometimes it is more efficient to have data and computation duplicated on processes
- Communication, Flexibility, Software Design
  - Competing for attention

Parallel Pi

- One processor collects NumInCircle and calculates Pi
- No need to agglomerate the calculation of localNumPoints
Assign Groups to Processors

- Tasks that can execute concurrently on different processes
- Tasks that communicate frequently on the same processors
- Resources may also direct decisions
  - Memory per cpu
  - Bandwidth

Parallel Pi

Functional Decomposition
The Diffusion Equation
Diffusion Equation

\[ \frac{\partial u}{\partial t} = D \left[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right] \]

- \( u(x,y,t) \): Temperature of surface
- \( D \): The diffusion coefficient
- Solve on a rectangular plane
  - Zero temp at boundary (Dirichlet)
  - Some initial temperature \( T \)
- Convenient because we know the solution
Numerical Solution

- Solve with finite difference, forward time centered
- Solution based on neighbors, previous time step

\[
\frac{u_{ij}^{n+1} - u_{ij}^n}{dt} = D \left[ \frac{u_{i+1,j}^n - 2u_{ij}^n + u_{i-1,j}^n}{dx^2} + \frac{u_{i,j+1}^n - 2u_{ij}^n + u_{i,j-1}^n}{dy^2} \right]
\]
Numerical Solution

- What the numerical solution is doing
  - A simple to understand approximation
    - Finite difference is discrete analog to the derivative
    - Approximate solution based on the solution of the neighbors from previous solution
Pseudo Code

solnData(size of grid)
oldSolnData(size of grid)

Calculate \( dt, dx, dy, \) coefficients, etc
Set initial conditions
Set boundary conditions
Loop over timesteps
  Loop over \( x \)
    Loop over \( y \)
      solve for \( \text{solnData}(\text{point}) \) from oldSoln
Profiling of the Serial Diffusion Equation

- Spends entire time in loop
- Data Dependency
  - One cell needs cell data from neighbors
- Granularity
  - Limited to every time step - fine
- Load Balancing
  - Equal work per cell
    - Equal cells per processor
Parallelize

- Partition Tasks
  - Init boundary conditions
  - Initial conditions
  - Calculate constants
  - Data structure split across processors
    - Loop limited to local sections of solution
Communication

- **Calculate constants**
  - Broadcast, or each proc does it

- **Boundary conditions**
  - None if the processors on boundaries calculate

- **Initial conditions**
  - None: we choose constant initial conditions

- **Solution data**
  - Must exchange the cells on borders of splits
Border Cells / Ghost Points

- When splitting up solnData, need data from other processors.
- Need a layer of cells from each processor
- Need to update each time step
Agglomerate

- Calculation of constants
  - Duplicate this on processors
    - It is cheap, the communication is not

- Boundary Conditions, Initial Conditions, Solution Data
  - These all operate on local section of an array
Mapping

- Very natural mapping
  - Divide the solution data (physical domain) as evenly as possible over processors.
  - Each agglomeration
    - One section of physical domain

- Topology
  - 0 1 2 3
  - 0 3 1 2
Parallel Pseudo Code

Determine numProcs
Divide grid evenly in x over numProcs
Calculate
local xStart,xStop
xLen = xStop-xStart
solnData(LocalGrid)
oldSolnData(LocalGrid)
Calculate
dx, dy, dt
Coefficients
If LocalGrid on Boundary
Init Boundary Conditions
do x=gc,xLen+gc
do y=gc,yGrid+gc
    initial con.
do timesteps
   do x=gc,xLen+gc
      do y=gc,yGrid+gc
          solve
          update ghost cells
          enddo
enddo
Initialization

- Completely new
- Discover
  - Size of job
  - Where this task is
- Fortran
  - Interfaces almost the same
  - No argc, argv
  - Add integer ierr

```c
/* Declare MPI status*/
MPI_Status status;

/* Initialize the MPI API: */
MPI_Init(&argc, &argv);

/* Request my ID number: */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

/* Total number of procs */
MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
```
Decomposition

- **Mynni**
  - Grid points in x
  - Divides over numProcs
- **Only one dimension of decomposition**

```c
/* Compute number of x-direction grid points allocated to me, * mynni */
mynniSum = nni + numProcs;
for ( k = 0; k < numProcs; k++ )
{
    mynni[k] = nni / numProcs;
    if ( k < (nni % numProcs) )
    {
        mynni[k]++;
    }
    if (myrank == 0)
        printf("mynni[%i] = %i\n", k, mynni[k]);
}
```
Topology

- **Topology**
  - Often tied to networking
- **Assumes linear assignment**
  - Potential error
  - Cartcreate, or, new communicator
- **Needed for ghostcells**

```c
/* Set the ranks of the processors to the left and right of me. These are the processors I will communicate with. */
rightProc = myrank + 1;
if(rightProc == numProcs)
    rightProc = MPI_PROC_NULL;
leftProc = myrank - 1;
if(leftProc == -1)
    leftProc = MPI_PROC_NULL;
```

```plaintext
L1
\_\_\_\_\_\_\_\_\_
    \_\_\_\_\_\_\_\_
        \_\_\_\_\_\_\_\_
            \_\_\_\_\_\_\_\_

L2
    \_\_\_\_\_\_\_\_\_
        \_\_\_\_\_\_\_\_\_
            \_\_\_\_\_\_\_\_\_
                \_\_\_\_\_\_\_\_\_
```

Jazz LCRC
Alternative Topology

- MPI Topology functions can do this for you
- Very useful for straightforward decompositions

```c
integer dims[NDIM];
Integer periods[NDIM];

err=MPI_Dims_create ( size, NUM_DIMS, dims );

err=MPI_Cart_create( 
    MPI_COMM_WORLD, 
    ndims, &dims, &periods, 
    reorder, &comm_cart );

for (i=0;i<NUM_DIMS;i++) {
    err=MPI_Cart_shift( 
        comm_cart, i, 1, 
        &source, &dest); 
}
```
Ghostcell Exchange

ODD:
- send to left; recv from left
- send to right; recv from right

EVEN:
- recv from right; send to right
- recv from left; send to left

MPI SEND(buffer, count, datatype, destination, tag, communicator)
MPI RECV(buffer, count, datatype, source, tag, communicator, status)
Ghostcell Exchange

0

1

2

3

MPI_Sendrecv(send to left, recv from right)

MPI_Sendrecv(send to right, recv from left)

Receives

Sends

Jazz LCRC
Finalize

- Understand the algorithm
  - Where can it be parallelized?
  - Where does it need it?
  - What packages have done it for me?
    - Time

- Planning the parallel implementation can save much time!
I/O

- Very important to think about
- Use of scientific data library
  - netCDF, pnetCDF
- Serial I/O
  - All data to a single processor
- Parallel I/O
  - All processes write to one or more files
Parallel I/O

- Every process writes out a file
  - Many files
  - Easier to write code
  - Post processing harder

- Logic to write out one file
  - More time, More communication
  - Library support: hdf5, pnetcdf, MPI I/O
Scientific Data Libraries

- Can mirror your data structures
  - Can reduce costly copies
  - Easy to understand the process
- Libraries allow self discovery
- Easy content browsing
- Portability
Data Library I/O Looks Like

Single Alg

Data Structures

File

Serial I/O for Parallel Alg.

0 1 2 3

Master

File

Parallel I/O for Parallel Alg.

0 1 2 3

File
Jumpshot

- MPE Application from MCS
- Illustrates the communication patterns and load balancing of application
- Understanding of applications
- Development of applications
Web Resources

- **Designing and Building Parallel Programs**

- **Tutorials from Maui High Performance Computing Center**
  - [http://www.mhpcc.edu/training/tutorials/Tutorials.html](http://www.mhpcc.edu/training/tutorials/Tutorials.html)

- **LLNL HPC Training**
  - [http://www.llnl.gov/computing/training/#workshops](http://www.llnl.gov/computing/training/#workshops)

- **Jazz Web Page**
  - [http://www.lcrc.anl.gov](http://www.lcrc.anl.gov)
Some Parallel Libraries

- Linear Algebra for Dense Systems
  - The BLAS, LAPACK, BLACS, PBLAS, ScaLAPACK

- Sparse Linear Algebra
  - The Sparse BLAS, The PIM Library

- Other Parallel Libraries
  - PESSL, NAG Parallel Libraries, PETSc
A Few Tools

- Profiling
  - Serial: gprof
  - Parallel: jumpshot

- Debugging
  - Serial: ddd, gdb,
  - Totalview

- Parallel Scientific applications on Jazz
  - Star-CD, IDL, MatLab, Gaussian98, Materials Studio (MSI)
  - Globus
  - Intel, Portland Group, Nag