Introduction to Parallel Computing
Why do Parallel Computing

• Limits of single CPU computing
  – performance
  – available memory

• Parallel computing allows one to:
  – solve problems that don’t fit on a single CPU
  – solve problems that can’t be solved in a reasonable time

• We can solve…
  – larger problems (Capability)
  – Faster/more cases (Capacity)
State-of-the-art HPC systems

Top System Performance [MFlop/s]

Year


IBM 704
IBM 709
IBM 7090
IBM 7030 "Stretch"
UNIVAC LARC
CDC 6600
CDC 7600
Burroughs ILLIAC IV
Cray-1
Cray-2/8
Cray X-MP/4
ETA 10-G/8
NEC SX-3/44R
M-13 (USSR)
CDC Cyber 205
Cray-100
Hitachi/Tsukuba CP-PACS/2048
Intel Paragon XP/S 140
Intel ASCI Red/9632
Intel ASCI Red/9152
Fujitsu Numerical Wind Tunnel
NEC Earth Simulator
SGI Project Columbia
IBM ASCI White
IBM RoadRunner
IBM Blue Gene/L
Cray Jaguar
IBM ASCI White
NEC Earth Simulator
SGI Project Columbia
IBM RoadRunner
IBM Blue Gene/L
Cray Jaguar
Outline

• Introduction to Parallel Computing
  – Performance and theoretical limits
  – Types of Parallel Computers
  – Programming Techniques

• Parallel Computing using MPI
  – Message Passing Model
  – Initializing and terminating programs
  – Point to point communications
  – Global Communications

• Overview

• Examples & Projects
Introduction to Parallel Computing
What is Parallel Computing

• Parallel computing: use of multiple processors or computers working together on a common task.
  – Each processor works on its section of the problem
  – Processors can exchange information
Performance Metrics

**Speedup**

\[ S_N = \frac{t_1}{t_N} \]

- \( t_1 \) is the serial execution time
- \( t_N \) is parallel execution time

**Efficiency**

\[ E_P = \frac{S_N}{N} \]

- \( N \) is the number of processes
Limits of Parallel Computing

• Theoretical Upper Limits
  – Amdahl’s Law

• Practical Limits
  – Load balancing
  – Non-computational sections

• Other Considerations
  – time to re-write code
Theoretical Upper Limits to Performance

- All parallel programs contain:
  - parallel sections (we hope!)
  - serial sections (unfortunately)
- Serial sections limit the parallel effectiveness
- Amdahl’s Law states this formally
Amdahl’s Law

- Amdahl’s Law places a strict limit on the speedup that can be realized by using multiple processors.
  - Run time
    \[ t_N = \left( \frac{f_p}{N} + f_s \right) t_1 \]
  - Speed up
    \[ S_N = \frac{1}{f_s + \frac{f_p}{N}} \]
  - Where
    - \( f_s \) = serial fraction of code
    - \( f_p \) = parallel fraction of code
    - \( N \) = number of processors
Practical Limits: Amdahl’s law vs. Reality

• In reality, the situation is even worse than predicted by Amdahl’s Law due to:
  – Load balancing (waiting)
  – Scheduling (shared processors or memory)
  – Communications
  – I/O

\[ f_p = 0.99 \]
Other Considerations

• Writing effective parallel applications is difficult!
  – Load balance is important
  – Communication can limit parallel efficiency
  – Serial time can dominate

• Is it worth your time to rewrite your application?
  – Do the CPU requirements justify parallelization?
  – Will the code be used just once?
Types of Parallel Computers

Flynn’s Taxonomy

- PC, Mainframes
- Vectorprocessors, Altivec, SSE3
- BlueGene, Cray XT3, Clusters, Grid

Single Instruction
- SISD
- SIMD

Multiple Instruction
- MISD
- MIMD

Memory Architecture

• Shared memory:
  – single address space. All processors have access to a pool of shared memory.
  – Examples: SGI Altix, IBM Power5 node

• Distributed memory:
  – each processor has its own local memory. Must do message passing to exchange data between processors.
  – Examples: Clusters
Programming Parallel Computers

• Programming single-processor systems is (relatively) easy due to:
  – single thread of execution
  – single address space

• Programming shared memory systems can benefit from the single address space

• Programming distributed memory systems is the most difficult due to multiple address spaces and need to access remote data

• Both parallel systems offer ability to perform independent operations on different data (MIMD) and implement task parallelism

• Both can be programmed in a data parallel, SIMD fashion
Problem Decomposition

- Domain Decomposition
  Each computing element executes the same task on different data

- Functional Decomposition
  Each computing element executes a different task on same or different/data

```plaintext
if CPU="a" then
  low=1; high=50
else if CPU="b" then
  low=51; high=101
endif
process( data[low:high] )

if CPU="a" then
  do_task_A()
else if CPU="b" then
  do_task_B()
endif
```
• **SPMD**: dominant programming model for shared and distributed memory machines.
  – One source code is written
  – Code can have conditional execution based on which processor is executing the copy
  – All copies of code are started simultaneously and communicate and sync with each other periodically
Shared Memory vs. Distributed Memory

• Tools can be developed to make any system appear to look like a different kind of system
  – distributed memory systems can be programmed as if they have shared memory, and vice versa
  – such tools do not produce the most efficient code, but might enable portability

• HOWEVER, the most natural way to program any machine is to use tools & languages that express the algorithm explicitly for the architecture.
Shared Memory Programming: OpenMP

• Shared memory systems (e.g. multi core workstations) have a single address space:
  – applications can be developed in which loop iterations (with no dependencies) are executed by different processors
  – shared memory codes are mostly data parallel, ‘SIMD’ kinds of codes
  – OpenMP is the new standard for shared memory programming (compiler directives)
  – Vendors offer native compiler directives (gcc, icc, xlc, gfortran, ifort, xlf, etc.)
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {

    int nthreads, tid;

    /* Fork a team of threads giving them their own copies of variables */
    #pragma omp parallel private(nthreads, tid)
    {
        /* Obtain thread number */
        tid =omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }

    /* All threads join master thread and disband */
}
Distributed Memory Programming: MPI

- Distributed memory systems have separate address spaces for each processor
  - Local memory accessed faster than remote memory
  - Data must be manually decomposed
  - MPI is the standard for distributed memory programming (library of subprogram calls)
  - Older message passing libraries include PVM and P4; all vendors have native libraries such as SHMEM (T3E) and LAPI (IBM)
Data Decomposition

- For distributed memory systems, the ‘whole’ grid or sum of particles is decomposed to the individual nodes:
  - Each node works on its section of the problem.
  - Nodes can exchange information using messages.

![Diagram of data decomposition](image)
Programming Multi-tiered Systems

• Systems with multiple shared memory nodes are becoming common for reasons of economics and engineering.

• Memory is shared at the node level, distributed above that:
  – Applications can be written using OpenMP + MPI
  – Developing apps with only MPI usually possible
Parallel Computing using MPI
Using MPI on local machine

• MPI development can be done on any single cpu machine

• Install one MPI flavor in your laptop/desktop
  – OpenMPI (*nix) http://www.open-mpi.org

• Develop and Debug locally
  – You can launch multiple processes in a single machine, emulating a large parallel environment

• Once finished debugging compile and deploy on the larger system for production runs
Message Passing Model

• Each process has its own local address space
• Processes communicate with each other through messages
• Advantages:
  – Works with both distributed and shared memory
  – Extremely general/portable; even allows for heterogeneous systems
• Disadvantages:
  – Communications take time, some problems difficult to parallelize
• **Message Passing Interface**
  
  – Library describing API for message passing on parallel computers: official bindings for Fortran, C and C++, many others available
  
  – Specification not implementation: many implementations available, either proprietary (e.g. AIX MP) or open source (OpenMPI, MPICH, etc.)
  
  – *De facto* standard for message passing, available on most systems, from desktops to supercomputers
Message Passing Programs

- MPI programs consist of multiple instances of a serial program that communicate by MPI library calls. These calls may be roughly divided into four classes:
  1. Calls used to initialize, manage, and finally terminate communications.
  2. Calls used to communicate between pairs of processors.
  3. Calls that perform communications operations among groups of processors.
  4. Calls used to create arbitrary data types.
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
    int rank, size;

    /* Initialize MPI */
    MPI_Init(&argc, &argv);
    /* Get total number of procs. */
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    /* Get my rank */
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    printf("I'm proc. %d of %d\n",rank,size);

    /* Close MPI */
    MPI_Finalize();
}
Compiling MPI programs

• MPI programs can be compiled using MPI wrapper compilers:

  $ mpicc HelloWorld.c -o HelloWorld

• MPI programs can be tested locally on desktop machines:

  $ mpirun -np 4 ./HelloWorld
  I’m proc. 1 of 4
  I’m proc. 2 of 4
  I’m proc. 3 of 4
  I’m proc. 0 of 4
MPI Messages

- MPI messages consist of 2 parts: the envelope and the message body:

<table>
<thead>
<tr>
<th>Envelope</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>buffer</td>
</tr>
<tr>
<td>destination</td>
<td>datatype</td>
</tr>
<tr>
<td>tag</td>
<td>count</td>
</tr>
<tr>
<td>communicator</td>
<td></td>
</tr>
</tbody>
</table>
Message Parameters

- **Communicator**
  - Defines group of processes. Usually `MPI_COMM_WORLD` that represents all processes

- **Datatype**
  - Allows for heterogeneous systems
  - User datatypes can also be defined (e.g. structures)

### MPI Datatype (C)
- **MPI_CHAR**: signed char
- **MPI_DOUBLE**: double
- **MPI_FLOAT**: float
- **MPI_INT**: int
- **MPI_LONG**: long
- **MPI_UNSIGNED**: unsigned int

### MPI Datatype (Fortran)
- **MPI_CHARACTER**: character
- **MPI_COMPLEX**: complex
- **MPI_DOUBLE_PRECISION**: double precision
- **MPI_INTEGER**: integer
- **MPI_LOGICAL**: logical
- **MPI_REAL**: real
Point-to-Point communication

Source

Message

Destination

int MPI_Send(void* buffer, int count, MPI_Datatype dtype, int destination, int tag, MPI_Comm comm)

int MPI_Recv(void *buffer, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Calling processes are blocked pending communication completion
program test_send_receive

  include 'mpif.h'
  integer :: tag, myrank, ierr, request, status(MPI_STATUS_SIZE)
  real, dimension(100) :: a

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

  tag = 10
  ! process 0 sends, process 1 receives
  if ( myrank == 0 ) then
    call MPI_SEND( a, 100, MPI_REAL, 1, tag, MPI_COMM_WORLD, ierr)
  else if ( myrank == 1 ) then
    call MPI_RECV( a, 100, MPI_REAL, 0, tag, MPI_COMM_WORLD, status, ierr)
  endif

  call MPI_FINALIZE(ierr)
A six-function version of MPI

- The following provide a complete specification of a message passing interface:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>Initialize MPI</td>
</tr>
<tr>
<td>MPI_Comm_Size</td>
<td>Find out how many processes there are</td>
</tr>
<tr>
<td>MPI_Comm_Rank</td>
<td>Find out which process I am</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>Send a message</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>Receive a message</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>Terminate MPI</td>
</tr>
</tbody>
</table>
Non-Blocking Communications

Program may proceed while waiting for communication
MPI_Isend / MPI_IRecv

• Post Sends/Recvs without blocking

```c
int MPI_Isend(void* buffer, int count, MPI_Datatype dtype, int destination, int tag, MPI_Comm comm, MPI_Request *request)
```

```c
int MPI_Irecv(void *buffer, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status, MPI_Request *request)
```

• Waiting and Testing for completion

```c
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

```c
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```
Example

program test_nonblocking

    include ‘mpif.h’
    integer :: tag1, tag2, myrank, ierr, request, status(MPI_STATUS_SIZE)
    real, dimension(100) :: a, b
    
call MPI_INIT(ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
    tag1 = 10; tag2 = 11

    if ( myrank == 0 ) then
        call MPI_IRECV( b, 100, MPI_REAL, 1, tag1, MPI_COMM_WORLD, request, ierr )
        call MPI_SEND( a, 100, MPI_REAL, 1, tag2, MPI_COMM_WORLD, ierr)
    else if ( myrank == 1 ) then
        call MPI_IRECV( b, 100, MPI_REAL, 0, tag2, MPI_COMM_WORLD, request, ierr )
        call MPI_SEND( a, 100, MPI_REAL, 0, tag1, MPI_COMM_WORLD, ierr)
    endif

    call MPI_WAIT( request, status, ierr )
    call MPI_FINALIZE(ierr)

end program test_nonblocking
Collective communications

• Main communication involving all processes in a group
  – Barrier synchronization across all processes
  – Broadcast data from 1 process to all processes
  – Gather data from all processes to 1 process
  – Global reduction operations such as sum, min, max, etc.
**Synchronization**

- **Calculation**
- **Barrier**
- **Calculation**

**Code Example**

```c
int MPI_Barrier(MPI_Comm comm)
```
Broadcast

• Broadcast a message from process root to all processes in group (itself included)

```
int MPI_Broadcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```
• Each process (root proc. included) sends a message to root process. Root process stores received data in rank order.

```
int MPI_Gather(void *sendbuffer, int sendcount, MPI_Datatype senddatatype, void* recvbuffer, int recvrcvcount, MPI_Datatype recvdatatype, int root, MPI_Comm comm)
```
Reduce

• Combines the data sent from each process using the selected operation and returns the result to process root.

```
int MPI_Reduce(void *sendbuffer, void* recvbuffer, int count,
               MPI_Datatype datatype, MPI_Op operation, int root, MPI_Comm comm)
```
Poisson Problem

\begin{align*}
\n\text{Interior:} & \quad \nabla^2 u = f(x, y) \\
\text{Boundary:} & \quad u(x, y) = g(x, y)
\end{align*}

Discretizing on a grid:

\begin{align*}
x_i &= \frac{i}{n+1}, \quad i = 0, \ldots, n + 1 \\
y_j &= \frac{j}{n+1}, \quad j = 0, \ldots, n + 1 \\
u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} &= f_{i,j} \\
h^2 &\quad (h \text{ is the grid spacing})
\end{align*}
Jacobi Iteration

Rearrange finite difference equation:

\[
  u_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j})
\]

Estimate \( u_{i,j}^{k+1} \) may be obtained from \( u_{i,j}^k \):

\[
  u_{i,j}^{k+1} = \frac{1}{4}(u_{i-1,j}^k + u_{i+1,j}^k + u_{i,j-1}^k + u_{i,j+1}^k - h^2 f_{i,j})
\]

```
real, dimension(1:nx,1:nx) :: unew,f
real, dimension(0:nx+1,0:nx+1) :: u
!
! jacob iteration
do i=1, nx
  do j=1, nx
    unew(i,j) = 0.25*(u(i-1,j)+u(i+1,j)+
                      u(i,j-1)+u(i,j+1)- &
                      h*h*f(i,j))
  enddo
enddo
```
Parallelization

- Use domain decomposition
- Most calculations use local data
- Cells along edge are required from other nodes
Use ghost/guard cells

Maintain local copy (●) of data belonging to neighbor (○)

- Before computation copy values from neighbor to guard cells
- All data required for calculation is now available locally

Messages

Calculation Cells

Guard Cells
Parallel Poisson Program

integer, parameter :: nprocs = 4
real, dimension( 1: nx/nprocs, 1:nx ) :: f, unew
real, dimension( 0:(nx/nprocs)+1, 0:nx+1 ) :: u

! Initialize MPI
call MPI_INIT(ierr)
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr)

!(...) 
do iter = 1, max_iter
   u(1:nx/nprocs,1:nx) = unew(1:nx/nprocs,1:nx)
      ! update guard cells
      ! (...)

! jacobi iteration
   do i = 1, nx/nprocs
      do j = 1, nx
         unew(i,j) = 0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)+&
                         u(i,j+1)- h*h*f(i,j))
      enddo
   enddo

   ! check convergence
   ! (...)
endo

! Close MPI
call MPI_FINALIZE(ierr)
integer :: l_rec_req, u_rec_req, l_send_req, u_send_req
integer, dimension( MPI_STATUS_SIZE ) :: status
real, dimension( 1:nx ) :: lbuf, ubuf

! Begin communication with lower neighbor
call MPI_IRECV( lbuf, nx, MPI_REAL, rank - 1, &
  10, MPI_COMM_WORLD, l_rec_req, ierr)
call MPI_ISEND( u(0,1:nx), nx, MPI_REAL, rank - 1, &
  10, MPI_COMM_WORLD, l_send_req, ierr)

! Begin communication with upper neighbor
call MPI_IRECV( ubuf, nx, MPI_REAL, rank + 1, &
  10, MPI_COMM_WORLD, u_rec_req, ierr)
call MPI_ISEND( u(nx/nprocs+1,1:nx), nx, MPI_REAL, rank + 1, &
  10, MPI_COMM_WORLD, &u_send_req, ierr)

! wait for send messages to complete
call MPI_WAIT( l_send_req, status, ierr )
call MPI_WAIT( u_send_req, status, ierr )

! wait for receive messages and copy guard cell values
call MPI_WAIT( l_rec_req, status, ierr )
u(0,1:nx) = lbuf
call MPI_WAIT( u_rec_req, status, ierr )
u(nx/nprocs+1, 1:nx) = ubuf
real :: diff, totdiff
!(...) ! calculate diff from local data
  diff = 0.0
  do i = 1, nx/nprocs
      do j = 1, nx
          diff = diff + ( u(i,j) - unew(i,j) ) ** 2
      enddo
  enddo
  ! sum diff from all nodes
  call MPI_REDUCE( diff, totdiff, 1, MPI_REAL, MPI_SUM, 0, &
                   MPI_COMM_WORLD, ierr)

  ! print result on node 0
  if ( rank == 0 ) then
      totdiff = sqrt( totdiff ) / h / (nx * nx )
      print *, "After", iter, " iterations eps = ", totdiff
  endif
Overview
Overview

- Parallel computing breaks the limits of single CPU computing in terms of performance and memory
- It can be applied from laptops to supercomputers
- MPI provides a standard/portable toolset for this computing paradigm
- Minimal learning curve / required hardware resources to begin parallel programming.
Parallelizing Programs

• New programming model required
  – Manual parallelization always beats automatic one

• Rethink algorithms
  – Fastest serial algorithm may not scale well in parallel
  – Take one step back and two forward

• Improve Parallel Performance
  – Load Balancing
  – Minimizing Communication
  – Overlapping Communication and Computation
Further Reading

- NCSA MPI Tutorial
- Using MPI: Portable Parallel Programming with the Message-passing Interface
  - W. Gropp, E. Lusk, A. Skjellum, 2nd Ed., MIT Press
- MPI: The complete reference, Vol 1, The MPI Core
- NCSA OpenMP Tutorial
- Parallel Programming in OpenMP
  - R. Chandra, etc., Kaufmann Ed.
- Using OpenMP
  - B. Chapman, etc., MIT Press
Examples & Projects
Example Programs I/II

• Source for examples can be found at
  – http://zamb.ist.utl.pt/mpi_tutorial/

• MPI Fundamentals
  – Minimal MPI code (mpi0.c/mpi0.f)
  – Local process number/ total number of processes (mpi1.c/mpi1.f)
  – An arbitrary number of processes can be launched on a single machine

• Message Passing basics
  – Blocking SEND/RECV (mpi2.c/mpi2.f)
  – Extend program so that node 0 initializes some data and node 1 verifies integrity
  – Experiment with other datatypes
Example Programs II/II

• Non blocking communications
  – Non-Blocking ISEND/IRECV (mpi3.c/mpi3.f)
  – Extend program so that nodes initializes local data and verify integrity of received message

• Collective communications
  – Broadcast example (mpi_bcast.c/mpi_bcast.f)
  – Gather example (mpi_gather.c/mpi_gather.f)
  – Reduce example (mpi_reduce.c/mpi_reduce.f)
Project 1

• Write a program that sends a token M times around a ring of N processes
  – Implement using blocking communications
  – Implement using non-blocking communications
Project 2

• Calculate $\pi$ in parallel using:

$$\pi = \int_0^1 \frac{4}{1 + x^2} \, dx$$

– Integrate using Euler’s method
– Split the integration interval over available nodes
– Each node integrates over its portion of interval
– Calculate final result using collective communications
Project 3

• Implement a program to measure communication bandwidth between two processes
  – Create a (large) array of some type
  – Use blocking point to point communication
  – Send a message from node 0 to node 1 and then send it back from node 1 to node 0
  – Measure time using MPI_Wtime()