CS4961 Parallel Programming

Lecture 16: Introduction to Message Passing

Mary Hall
October 29, 2009

Administrative

- Homework assignment 3 will be posted today (after class)
- Due, Thursday, November 5 before class
  - Use the "handin" program on the CADE machines
  - Use the following command:

    "handin cs4961 hw3 <gzipped tar file>"

NEW: VTUNE PORTION IS EXTRA CREDIT!

- Mailing list set up: cs4961@list.eng.utah.edu
- Next week we'll start discussing final project
  - Optional CUDA or MPI programming assignment part of this

A Few Words About Final Project

- Purpose:
  - A chance to dig in deeper into a parallel programming model and explore concepts.
  - Present results to work on communication of technical ideas
- Write a non-trivial parallel program that combines two parallel programming languages/models. In some cases, just do two separate implementations.
  - OpenMP + SSE-3
  - OpenMP + CUDA (but need to do this in separate parts of the code)
  - TBB + SSE-3
  - MPI + OpenMP
  - MPI + SSE-3
  - MPI + CUDA
- Present results in a poster session on the last day of class

Example Projects

- Look in the textbook or on-line
  - Recall Red/Blue from Ch. 4
    - Implement in MPI (+ SSE-3)
    - Implement main computation in CUDA
  - Algorithms from Ch. 5
  - SOR from Ch. 7
    - CUDA implementation?
  - FFT from Ch. 10
  - Jacobi from Ch. 10
  - Graph algorithms
  - Image and signal processing algorithms
  - Other domains…

Today's Lecture

- Message Passing, largely for distributed memory
- Message Passing Interface (MPI): a Local View language
- Sources for this lecture
  - Larry Snyder, http://www.cs.washington.edu/education/courses/524/08wi/
  - Online MPI tutorial http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html

Message Passing

- Message passing is the principle alternative to shared memory parallel programming
  - Based on Single Program, Multiple Data (SPMD)
  - Model with send() and recv() primitives
  - Message passing is universal, but low-level
  - More even than threading, message passing is locally focused -- what does each processor do?
  - Isolation of separate address spaces
    - + no data races
    - + forces programmer to think about locality, so good for performance
    - + architecture model exposed, so good for performance
    - - low level
    - - complexity
    - - code growth!

Message Passing Libraries (1)

- Many "message passing libraries" were once available
  - Chameleon, from ANL.
  - CMMD, from Thinking Machines.
  - Express, commercial.
  - MPL, native library on IBM SP-2.
  - NX, native library on Intel Paragon.
  - Zipcode, from LLL.
  - PVM, Parallel Virtual Machine, public, from ORNL/UTK.
  - Others...
  - MPI, Message Passing Interface, now the industry standard.
- Need standards to write portable code.

Message Passing Libraries (2)

- All communication, synchronization require subroutine calls
  - No shared variables
  - Program run on a single processor just like any uniprocessor program, except for calls to message passing library
- Subroutines for
  - Communication
    - Pairwise or point-to-point: Send and Receive
    - Collectives all processor get together to
      - Move data: Broadcast, Scatter/gather
      - Compute and move: sum, product, max, … of data on many processors
  - Synchronization
    - Barrier
    - No locks because there are no shared variables to protect
  - Queries
    - How many processes? Which one am I? Any messages waiting?

Novel Features of MPI

- Communicators encapsulate communication spaces for library safety
- Datatypes reduce copying costs and permit heterogeneity
- Multiple communication modes allow precise buffer management
- Extensive collective operations for scalable global communication
- Process topologies permit efficient process placement, user views of process layout
- Profiling interface encourages portable tools

Slide source: Bill Gropp, ANL

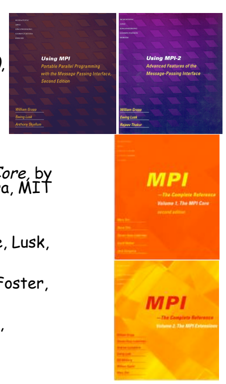
MPI References

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

Slide source: Bill Gropp, ANL

Books on MPI

- *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (2nd edition), by Gropp, Lusk, and Skjellum, MIT Press, 1999.
- *Using MPI-2: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Thakur, MIT Press, 1999.
- *MPI: The Complete Reference - Vol 1 The MPI Core*, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1998.
- *MPI: The Complete Reference - Vol 2 The MPI Extensions*, by Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir, MIT Press, 1998.
- *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
- *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.



Slide source: Bill Gropp, ANL

Working through an example

- We'll write some message-passing pseudo code for Count3 (from Lecture 4)

```
int array[length];                              The data is global
int t;                                          Number of desired threads
int total;                                      Result of computation, grand total
forall(j in(0..t-1))
{
  int size=mySize(array,0);                     Figure size of local part of global data
  int myData[size]=localize(array[]);
                                                Associate my part of global data with
                                                local variable
  int i, priv_count=0;                          Local accumulation
  for(i=0; i<size; i++)
  {
    if(myData[i]==3)
    {
      priv_count++;
    }
  }
  total =+/priv_count;                          ompute grand total
}
```

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this 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*, a number between 0 and size-1, identifying the calling process

Slide source: Bill Gropp

Hello (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;
}
```

Slide source: Bill Gropp

Hello (Fortran)

```fortran
program main
include 'mpif.h'
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

Slide source: Bill Gropp

Hello (C++)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "I am " << rank << " of " << size <<
            "\n";
    MPI::Finalize();
    return 0;
}
```

Slide source: Bill Gropp

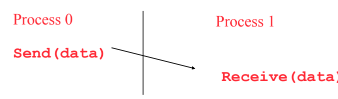
Notes on Hello World

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the `printf/print` statements
- I/O not part of MPI-1 but is in MPI-2
  - print and write to standard output or error not part of either MPI-1 or MPI-2
  - output order is undefined (may be interleaved by character, line, or blocks of characters),
- The MPI-1 Standard does not specify how to run an MPI program, but many implementations provide `mpirun -np 4 a.out`

Slide source: Bill Gropp

MPI Basic Send/Receive

- We need to fill in the details in



- 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?

Slide source: Bill Gropp, ANL

Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
  - Provides necessary support for libraries
- A group and context together form a communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called **MPI_COMM_WORLD**

Slide source: Bill Gropp

MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - 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, in particular ones for subarrays

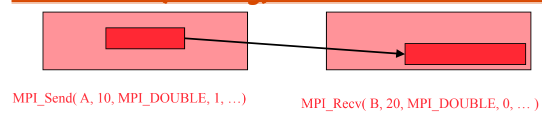
Slide source: Bill Gropp

MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes

Slide source: Bill Gropp

MPI Basic (Blocking) Send

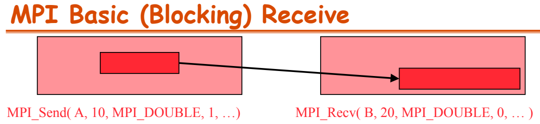


MPI_SEND(start, count, datatype, dest, tag, comm)

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

Slide source: Bill Gropp

MPI Basic (Blocking) Receive



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

- Waits until a matching (both **`source`** and **`tag`**) message is received from the system, and the buffer can be used
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**
- **`tag`** is a tag to be matched on or **`MPI_ANY_TAG`**
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error
- **`status`** contains further information (e.g. size of message)

Slide source: Bill Gropp, ANL

A Simple MPI Program

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  /* Process 0 sends and Process 1 receives */
  if (rank == 0) {
    buf = 123456;
    MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              &status );
    printf( "Received %d\n", buf );
  }

  MPI_Finalize();
  return 0;
}
```

Slide source: Bill Gropp, ANL

Figure 7.1 An MPI solution to the Count 3s problem.

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

int main(argc, argv)
int argc;
char **argv;
{
  int myID, value, numProcs;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myID);

  length_per_process=length/numProcs;
  myArray=(int *) malloc(length_per_process*sizeof(int));

  /* Read the data, distribute it among the various processes */
  if(myID==RootProcess)
  {
    if((fp=fopen(*argv, "r"))==NULL )
    {
      printf("fopen failed on %s\n", filename);
      exit(0);
    }
    fscanf(fp,"%d", &length);     /* read input size */

    for(p=0; p<numProcs-1; p++)   /* read data on behalf of each */
    {                             /* of the other processes */
      for(i=0; i<length_per_process; i++)
      {
        fscanf(fp,"%d", myArray+i);
      }
      MPI_Send(myArray, length_per_process, MPI_INT, p+1,
               tag, MPI_COMM_WORLD);
```


Figure 7.1 An MPI solution to the Count 3s problem. *(cont.)*

```
    }

    for(i=0; i<length_per_process; i++)  /* Now read my data */
    {
      fscanf(fp,"%d", myArray+i);
    }
  }
  else
  {
    MPI_Recv(myArray, length_per_process, MPI_INT, RootProcess,
             tag, MPI_COMM_WORLD, &status);
  }

  /* Do the actual work */
  for(i=0; i<length_per_process; i++)
  {
    if(myArray[i]==3)
    {
      myCount++;                           /* Update local count */
    }
  }

  MPI_Reduce(&myCount,&globalCount, 1, MPI_INT, MPI_SUM,
             RootProcess, MPI_COMM_WORLD);

  if(myID==RootProcess)
  {
    printf("Number of 3's: %d\n", globalCount);
  }
  MPI_Finalize()
  return 0;
}
```


Code Spec 7.8 `MPI_Scatter()`.

`MPI_Scatter()`

```
int MPI_Scatter(                        // Scatter routine
  void          *sendbuffer,            // Address of the data to send
  int           sendcount,              // Number of data elements to send
  MPI_Datatype  sendtype,               // Type of data elements to send
  int           destbuffer,             // Address of buffer to receive data
  int           destcount,              // Number of data elements to receive
  MPI_Datatype  desttype,               // Type of data elements to receive
  int           root,                   // Rank of the root process
  MPI_Comm      *comm                   // An MPI communicator
);
```

Arguments:

- The first three arguments specify the address, size, and type of the data elements to send to each process. These arguments only have meaning for the root process.
- The second three arguments specify the address, size, and type of the data elements for each receiving process. The size and type of the sending data and the receiving data may differ as a means of converting data types.

(continued)

Code Spec 7.8 `MPI_Scatter()`. *(cont.)*

- The seventh argument specifies the root process that is the source of the data.
- The eighth argument specifies the MPI communicator to use.

Notes:

This routine distributes data from the root process to all other processes, including the root. A more sophisticated version of the routine, `MPI_Scatterv()`, allows the root process to send different amounts of data to the various processes. Details can be found in the MPI standard.

Return value:

An MPI error code.

Figure 7.2

Replacement code (for lines 16–48 of Figure 7.1) to distribute data using a scatter operation.

```
length_per_process=length/size;
myArray=(int *) malloc(length_per_process*sizeof(int));

array=(int *) malloc(length*sizeof(int));

/* Read the data, distribute it among the various processes */
if(myID==RootProcess)
{
  if((fp=fopen(*argv, "r"))==NULL)
  {
    printf("fopen failed on %s\n", filename);
    exit(0);
  }
  fscanf(fp,"%d", &length);       /* read input size */

  for(i=0; i<length-1; i++)       /* read entire input file */
  {
    fscanf(fp,"%d", myArray+i);
  }
}

MPI_Scatter(Array, length_per_process, MPI_INT,
            myArray, length_per_process, MPI_INT,
            RootProcess, MPI_COMM_WORLD);
```


Other Basic Features of MPI

- MPI_Gather
  - Analogous to MPI_Scatter
- Scans and reductions
- Groups, communicators, tags
  - Mechanisms for identifying which processes participate in a communication
- MPI_Bcast
  - Broadcast to all other processes in a "group"

Figure 7.4 Example of collective communication within a group.

```
int numCols;                /* initialized elsewhere */

void broadcast_example()
{
  int **ranks;              /* the ranks that belong to each group */
  int myRank;
  int rowNumber;            /* row number of this process */
  int random;               /* value that we would like to broadcast */
  rowNumber=myRank/numCols;
  MPI_Group globalGroup, newGroup;
  MPI_Comm rowComm[numCols];

  /* initialize ranks[][] array */
  ranks[0]={0,1,2,3};  /* not legal C */
  ranks[1]={4,5,6,7};
  ranks[2]={8,9,10,11};
  ranks[3]={12,13,14,15};

  /* Extract the original group handle */
  MPI_Comm_group(MPI_COMM_WORLD, &globalGroup);

  /* Define the new group */
  MPI_Group_incl(globalGroup, P/numCols, ranks[rowNumber], &newGroup);

  /* Create new communicator */
  MPI_Comm_create(MPI_COMM_WORLD, newGroup, &newComm);

  random=rand();

  /* Broadcast 'random' across rows */
  MPI_Bcast(&random, 1, MPI_, rowNumber*numCols, newComm);
}
```


Figure 7.5 A 2D relaxation replaces—on each iteration—all interior values by the average of their four nearest neighbors.

1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0

Interior value
Boundary value

Figure 7.6 MPI code for the main loop of the 2D SOR computation.

```
#define Top      0
#define Left     0
#define Right    (Cols-1)
#define Bottom   (Rows-1)

#define NorthPE(i)      ((i)-Cols)
#define SouthPE(i)      ((i)+Cols)
#define EastPE(i)       ((i)+1)
#define WestPE(i)       ((i)-1)
...
do
{ /*
   * Send data to four neighbors
   */
  if(row !=Top)                                 /* Send North */
  {
    MPI_Send(&val[1][1], Width-2, MPI_FLOAT,
             NorthPE(myID), tag, MPI_COMM_WORLD);
  }

  if(col !=Right)                               /* Send East */
  {
    for(i=1; i<Height-1; i++)
    {
      buffer[i-1]=val[i][Width-2];
    }
    MPI_Send(buffer, Height-2, MPI_FLOAT,
             EastPE(myID), tag, MPI_COMM_WORLD);
  }

  if(row !=Bottom)                              /* Send South */
  {
    MPI_Send(&val[Height-2][1], Width-2, MPI_FLOAT,
            SouthPE(myID), tag, MPI_COMM_WORLD);
  }

  if(col !=Left)                                /* Send West */
  {
```


Figure 7.6 MPI code for the main loop of the 2D SOR computation. *(cont.)*

```
    for(i=1; i<Height-1; i++)
    {
      buffer[i-1]=val[i][1];
    }
    MPI_Send(buffer, Height-2, MPI_FLOAT,
             WestPE(myID), tag, MPI_COMM_WORLD);
  }

  /*
   * Receive messages
   */
  if(row !=Top)                           /* Receive from North */
  {
    MPI_Recv(&val[0][1], Width-2, MPI_FLOAT,
             NorthPE(myID), tag, MPI_COMM_WORLD, &status);
  }

  if(col !=Right)                         /* Receive from East */
  {
    MPI_Recv(&buffer, Height-2, MPI_FLOAT,
             EastPE(myID), tag, MPI_COMM_WORLD, &status);
    for(i=1; i<Height-1; i++)
    {
      val[i][Width-1]=buffer[i-1];
    }
  }

  if(row !=Bottom)                        /* Receive from South */
  {
    MPI_Recv(&val[0][Height-1], Width-2, MPI_FLOAT,
             SouthPE(myID), tag, MPI_COMM_WORLD, &status);
  }

  if(col !=Left)                          /* Receive from West */
  {
    MPI_Recv(&buffer, Height-2, MPI_FLOAT,
             WestPE(myID), tag, MPI_COMM_WORLD, &status);
    for(i=1; i<Height-1; i++)
    {
      val[i][0]=buffer[i-1];
    }
  }

  delta=0.0; /* Calculate average, delta for all points */
  for(i=1; i<Height-1; i++)
  {
    for(j=1; j<Width-1; j++)
    {
```


Figure 7.6 MPI code for the main loop of the 2D SOR computation. *(cont.)*

```
      average=(val[i-1][j]+val[i][j+1]+
               val[i+1][j]+val[i][j-1])/4;
      delta=Max(delta, Abs(average-val[i][j]));
      new[i][j]=average;
    }
  }

  /* Find maximum diff */
  MPI_Reduce(&delta, &globalDelta, 1, MPI_FLOAT, MPI_MIN,
             RootProcess, MPI_COMM_WORLD);
  Swap(val, new);
} while(globalDelta < THRESHOLD);
```


MPI Critique (Snyder)

- Message passing is a very simple model
- Extremely low level; heavy weight
  - Expense comes from λ and lots of local code
  - Communication code is often more than half
  - Tough to make adaptable and flexible
  - Tough to get right and know it
  - Tough to make perform in some (Snyder says most) cases
- Programming model of choice for scalability
- Widespread adoption due to portability, although not completely true in practice