Introduction to MPI

CDP
Shared Memory vs. Message Passing

**Shared Memory**
- Implicit communication via memory operations (load/store/lock)
- Global address space

**Message Passing**
- Communicate data among a set of processors without the need for a global memory
- Each process has its own local memory and communicated with others using messages
Shared Memory

• Advantages
  • **User Friendly:** global address space provides a user-friendly programming
  • **Fast and Uniform:** data sharing between tasks is both fast and uniform due to proximity of memory to CPUs

• Disadvantages
  • **Scalability:** primary disadvantage is the lack of scalability between memory and CPUs.
    Adding more CPUs can increase traffic on the shared memory CPU path.
  • **Responsibility:** Programmer responsibility for synchronization constructs that ensure “correct” access of global memory
  • **Hardware:** it becomes increasingly difficult and expensive to design and produce a shared memory machines with increasing number of processors
Message Passing

• Advantages
  • **Scalability**: adding more CPUs won’t harm CPU-memory bandwidth
  • **Responsibility**: elimination of the need for synchronization constructs such as semaphores, monitors, etc...
  • **Distributed**: naturally supports distributed computation

• Disadvantages
  • **Copy Overhead**: data exchanged among processors cannot be shared; it is rather copied (using send/receive messages, not without a cost)
  • **Complicated**: less natural transition from serial implementation
Overview - What is MPI?

Message Passing Interface

• MPI is a message-passing library
• Industry Standard
  • Developed by a consortium of corporations, government labs and universities
  • "Standard" by consensus of MPI Forum participants from over 40 organizations
• The first standard and portable message passing library with good performance
• MPI consists of 128 functions for
  • Point-to-Point message passing
  • User defined datatypes
  • Collective communication
  • Communicator and group management
  • Process topologies
  • Environmental management
• MPI v2 is now becoming the standard
  • Extends (does not change) MPI
What does MPI offer?

• **Standardization**
  • Rely on your MPI code to execute under any MPI implementation running on your architecture.

• **Portability**
  • Designed to supports most environments; very low resource requirements
  • Today your code is parallel; tomorrow it is distributed

• **Performance**
  • Meet industry’s performance demands

• **Richness**
  • 128 functions that allows many different communication methods
What is missing in MPI?

• **Dynamic process management**
  • All the processes created at initiation; cannot be changed.
  • MPI v2 already support dynamic processes

• **Shared memory operations**
  • Share data only via message passing

• **Multi-threading issues**
  • Threads are not supported by MPI (no shared memory)
  • Can use OpenMP with MPI

• **C++ bindings**
  • Can use from C++ code but does not support exceptions, namespaces, classes, etc.
  • MPI v2 already supports C++
Design and Implement an MPI Program

• **Serial**
  • When possible, start with a debugged serial version
  • Much easier to debug when running serial

• **Design**
  • Design parallel algorithm

• **Implement**
  • Write code, making calls to MPI library
  • Compile

• **Start Slow**
  • Run with a few nodes first, increase number gradually
  • Easier to debug with small amount of processes
Basic Outline of an MPI Program

• **Initialization**
  • Initialize communications

• **Algorithm**
  • Communicate to share data between processes
  • The logic of your program

• **Finalize**
  • Exit in a "clean" fashion from the message-passing system when done communicating
Format of MPI routines

• **C bindings:**
  • \( rc = \text{MPI}_\text{xxxx} \)(parameter, ... )
  • \( rc \) is error code, = \text{MPI\_SUCCESS} if successful

• All MPI routines for point-to-point communication and collective communication have integer return type.

• Header file required
  • \#include "mpi.h"
  • for C programs
6 Basic MPI calls

- MPI_Init
- MPI_Finalize
- MPI_Comm_rank
- MPI_Comm_size
- MPI_Send
- MPI_Recv
Initializing an MPI process

• **MPI_Init**
  • Initialize environment for communication
  • The first MPI call in any MPI process
  • One and only one call to **MPI_Init** per process

• **C syntax**
  • `int MPI_Init(int *argc, char ***argv)`

• Process creation is done by the call to
  • `mpiexec/mpirun -np <num_processes> <executable>`
Exiting from MPI

• **MPI_Finalize**
  • Exit in a "clean" fashion when done communicating
  • Cleans up state of MPI.
  • The last call of an MPI process
  • Must be called only when there is no more pending communications

• **C syntax**
  • `int MPI_Finalize()"
MPI Skeleton Program

#include "mpi.h"
int main(int argc, char* argv[]) {
    int rc = MPI_Init(&argc, &argv);
    if (rc != MPI_SUCCESS) {
        printf ("Error starting MPI program."
                " Terminating.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }
    printf ("Hello world\n");
    MPI_Finalize();
    return 0;
}
Basic MPI Definitions

• Group
  • An ordered set of processes
  • Has its own unique identifier (handle)
    • Assigned by the system
    • Unknown to the user
  • Associated with a communicator
  • Initially, all processes are members of the group given by the predefined communicator `MPI_COMM_WORLD`

• Rank
  • Unique, integer identifier for a process within a group
  • Sometimes called a "process ID"
  • Contiguous and begin at zero
  • Used to specify the source and destination of messages
Communicator

• Defines the collection of processes (group) which may communicate with each other (context)
• Possesses its own unique identifier (handle)
• Most MPI subroutines require you to specify the communicator as an argument
• We can create and remove groups/communicators during the program runtime
• **MPI_COMM_WORLD** is the predefined communicator which includes all processes in the MPI application
Rank and size within a communicator

• **MPI_Comm_rank**
  • Gets a process' rank within a communicator
  • `int MPI_Comm_rank(MPI_Comm comm, int *rank)`

• **MPI_Comm_size**
  • Gets the number of processes within a communicator
  • `int MPI_Comm_size(MPI_Comm comm, int *size)`
MPI Communicator Rank/Size Example

```c
#include "mpi.h"
int main(int argc, char* argv[]) { 
    int numtasks, rank, rc;
    rc = MPI_Init(&argc, &argv);
    if (rc != MPI_SUCCESS) {
        printf("Error starting MPI program. Terminating.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Number of tasks= %d My rank= %d\n", numtasks, rank);
    /*** Do some useful work ***/ 
    MPI_Finalize(); 
    return 0;
}
```
Sending and receiving messages

• **MPI_Send**
  - Basic blocking send operation
  - Called "standard" send mode
  - int `MPI_Send(void *sndbuf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`

• **MPI_Recv**
  - Basic blocking receive operation
  - int `MPI_Recv(void *recvbuf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
MPI messages

- **Message** = data + envelope

\[ \text{MPI\_Send (startbuf, count, datatype, dest, tag, comm)} \]
Data

- **startbuf**: starting location of data
- **count**: number of elements
  - receiver >= sender
  - Otherwise, will return **MPI_ERR_TRUNCATE** and will update only the lower elements on the receiver
- **datatype**: basic or derived
  - receiver == sender

\[
\text{MPI\_Send} (\text{startbuf}, \text{count}, \text{datatype}, \text{dest}, \text{tag}, \text{comm})
\]
MPI datatypes for C

• Derived datatypes
  • Mixed datatypes
  • Non-contiguous data

• When should we use each?
  • Mixed datatypes when data is known during compilation
  • MPI_Packed when data types are dynamic

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>uninterpreted 8-bit value</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>data packed by MPI_Pack()</td>
</tr>
</tbody>
</table>
Envelope

- **dest**: Destination or source
  - Rank in a communicator of sender/receiver respectively
  - Must match or receiver may use `MPI_ANY_SOURCE`

- **tag**: Message identifier
  - Integer chosen by programmer
  - Must match or receiver may use `MPI_ANY_TAG`

- **comm**: Communicator
  - Defines communication group
  - Receiver's and Sender's arguments must be the same; no wildcard

```c
MPI_Send(startbuf, count, datatype, dest, tag, comm)
```
Send/Recv Simple Example

Process 0 sends a message to process 1.

```c
char msg[20];
int myrank, tag=99;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find my rank */
if (myrank==0) {
    strcpy(msg, "Hello there");
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag,
             MPI_COMM_WORLD);
} else {
    MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
              &status);
}
...```
Multiplying a Dense Matrix with a Vector

• Reminder
  • The computation of each cell in the output vector is independent of the others
  • We can divide the output cells between the processes
#include "mpi.h"

#define SIZE_TAG 1
#define MATRIX_TAG 2
#define VECTOR_TAG 3
#define BALANCE_TAG 4
#define RES_VECTOR_TAG 5

int main(int argc, char* argv[]) {
    int numtasks, rank, rc;
    MPI_Status status;

    rc = MPI_Init(&argc, &argv);
    if (rc != MPI_SUCCESS) {
        printf("Error starting MPI program. Terminating.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    ...

    Tags for send/recv. We will use it later.
unsigned int size;
int** matrix;
int* vector;
int* res_vector;

if (rank == 0) {
    // Main process allocates and fill the matrix and vector
    get_matrix_and_vector(&matrix, &vector, &size);

    // Main process sends the matrix and vector to the others
    for(int i=1; i < numtasks; i++) {
        MPI_Send(&size, 1, MPI_UNSIGNED, i, SIZE_TAG, MPI_COMM_WORLD);
        MPI_Send(matrix, size*size, MPI_INT, i, MATRIX_TAG, MPI_COMM_WORLD);
        MPI_Send(vector, size, MPI_INT, i, VECTOR_TAG, MPI_COMM_WORLD);
    }
} else {
    // Other process receive the matrix and vector from the main process
    MPI_Recv(&size, 1, MPI_UNSIGNED, 0, SIZE_TAG, MPI_COMM_WORLD, &status);
    matrix = (int**) malloc(size*size*sizeof(int));
    vector = (int*) malloc(size*sizeof(int));

    MPI_Recv(matrix, size*size, MPI_INT, 0, MATRIX_TAG, MPI_COMM_WORLD, &status);
    MPI_Recv(vector, size, MPI_INT, 0, VECTOR_TAG, MPI_COMM_WORLD, &status);
}
int from_line, to_line;
// Each process calculates the result vector lines to work on
lines_balance(&from_line,&to_line, rank, numtasks, size);
res_vector = (int*) malloc(size*sizeof(int));

// All the processes calculate the result
for (i=from_line; i <= to_line; i++) {
    res_vector[i] = 0;
    for (j=0; j < size; j++) {
        res_vector[i] += matrix[i][j]*vector[j];
    }
}

...
```c
int buff[2];
if (rank != 0) {
    // Each process sends to main process the result vector
    buff[0] = from_line;
    buff[1] = to_line;
    MPI_Send(buff, 2, MPI_INT, 0, BALANCE_TAG, MPI_COMM_WORLD);
    MPI_Send(res_vector+from_line, to_line-from_line+1, MPI_INT, 0, RES_VECTOR_TAG, MPI_COMM_WORLD);
} else {
    // Main process receives the result from all the others
    for(int i=1; i < numtasks; i++) {
        MPI_Recv(buff, 2, MPI_INT, i, BALANCE_TAG, MPI_COMM_WORLD, &status);
        MPI_Recv(res_vector+buff[0], buff[1]-buff[0]+1, MPI_INT, i, VECTOR_TAG, MPI_COMM_WORLD, &status);
    }
}

/** Result vector is ready on the main process **/
MPI_Finalize();
free(matrix); free(vector); free(res_vector);
return 0;
```