

# MPI Programming

Sanjay Rajopadhye  
Colorado State University  
Fall 2011 Week 9

## Objectives

- Distributed memory programming
- Message passing (MPI) API
- Understanding MPI execution model
- Basic MPI functions
- Lab: first MPI program



## Outline

- Message Passing Model: a simple tweak on OpenMP and shared memory programming
- MPI programs
  - Coding
  - Compiling
  - Running
  - Benchmarking



## MPI Fundamentals

- MPI is just like OpenMP except
  - all variables are private (no shared variables)
  - plus some additional (communication) libraries
- SPMD (Single-Program-Multiple-Data) programming style

## Flynn's Taxonomy [1972]

- Parallel Hardware Classification

<b>SISD</b> Single Instruction Single Data	<b>SIMD</b> Single Instruction Multiple Data
<b>MISD</b> Multiple Instruction Single Data	<b>MIMD</b> Multiple Instruction Multiple Data

## SPMD Programming

- User writes a single program
- Multiple “instances” execute, on (potentially) different data sets
- The program is parameterized so that each process works on its “own” subset of data
- Two special functions à la `get_num_threads` and `get_thread_num`



## Processes

- Number of processes is fixed at start of program (load time)
- Remains fixed throughout execution
- Each process has a unique id
- Performs one of two things:
  - computation (on local data)
  - communication with other processes



## MPI Advantages

- Portable: currently the most widely used library for parallel programming
- High performance
- Intended (originally) for distributed memory parallel computers
- Allows explicit control over communication/ memory hierarchy

## History

- Late 1980s:
  - many distributed memory parallel machines, each vendor had unique libraries
- 1989: PVM (Parallel Virtual Machine) developed at ORNL
- 1992-97: MPI Standardization
  - version 1.0 in 1994, version 2.0 in 1997
- Stable since then (version 2.2 in 2009)

## Essential MPI

- Basic library functions:
  - **MPI\_Init** Initialize MPI
  - **MPI\_Finalize** Shut down
  - **MPI\_Comm\_rank** Who am I?
  - **MPI\_Comm\_size** How many of us are there?
  - **MPI\_Barrier** Synchronize with everyone
  - **MPI\_Wtime** Measure time elapsed
  - **MPI\_Wtick** Find the accuracy of the timer

## My first MPI Program

- Contrived example (as usual)
- Pleasantly (not embarrassingly) parallel
- Boolean Circuit Satisfiability
  - Given an  $n$ -input Boolean circuit, what subset of the  $2^n$  input combinations produces a result 1.
  - NP-complete: enumeration is the best known way to solve the problem (without approximations)
  - But all candidates can be tested independently

## Circuit Satisfiability

- General problem: solve for any given circuit
- Our example: “hardwire” the circuit (Fig 4.2)
  - 16 inputs (64k combinations) encode the 16-bit input pattern as an integer.
- Define a function **check\_circuit** function that takes an integer as argument and if the circuit evaluates to a 1 on that bit pattern, then prints the input combination
- Call this function in a loop (64k times)

```

#define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)
                        /*Returns 1 if 'i'th bit of 'n' is 1 */

void check_circuit (int id, int z) {
    int v[16];          /* Each element is a bit of z */
    int i;
    for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(z,i);

    if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
        && (!v[3] || !v[4]) && (v[4] || !v[5])
        && (v[5] || !v[6]) && (v[5] || v[6])
        && (v[6] || !v[15]) && (v[7] || !v[8])
        && (!v[7] || !v[13]) && (v[8] || v[9])
        && (v[8] || !v[9]) && (!v[9] || !v[10])
        && (v[9] || v[11]) && (v[10] || v[11])
        && (v[12] || v[13]) && (v[13] || !v[14])
        && (v[14] || v[15])) {
        printf ("%d) %d%d%d%d%d%d%d%d%d%d%d%d%d%d\n",
            id, v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9],v[10],v[11],v[12],
            v[13],v[14],v[15]);
        fflush (stdout);
    }
}

```

## Main (sequential) program

```

for (i = 0; i < 65536; i ++) check_circuit (0, i);

```

## Parallelization

- Each iteration of the loop is an independent piece of work
- Iterations allocated in a cyclic manner to the processes
- Each process does the check on its allocated input combinations
- If it finds a satisfiable combination it prints it.

## Local variables

```
int main (int argc, char *argv[]) {  
    int i;  
    int id; /* Process rank */  
    int p; /* Number of processes */  
    void check_circuit (int, int);
```

- Include `argc` and `argv`: used to initialize MPI
- One copy of each variable in every process

## Initialization

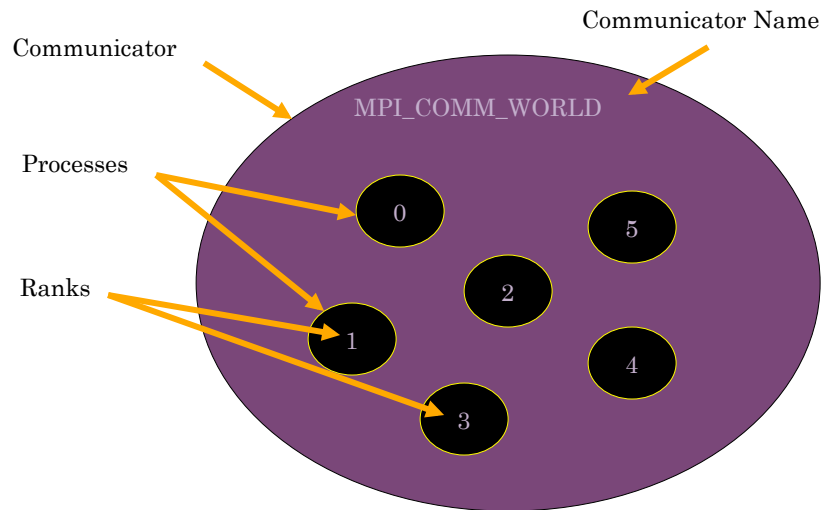
### **MPI\_Init (&argc, &argv);**

- First MPI function called by a process
- Not necessarily the first executable statement
- System does the setup:
  - copy the declared variables into local address space of each process
  - create “Communicators”

## Communicators

- Opaque object that determines the “communication universe” in which a point-to-point or collective operation is to operate.
- Simplified view:
  - MPI allows grouping processes so that they may communicate amongst themselves
  - Communicators are the names of such groups
  - default communicator **MPI\_COMM\_WORLD**

## Communicators



## How many processes

**`MPI_Comm_size (MPI_COMM_WORLD, &p);`**

- First argument is communicator
- Second argument is the address of the result (must be an integer)

## Which process

```
MPI_Comm_rank (MPI_COMM_WORLD, &myid);
```

- First argument is communicator
- Second argument is the address of the result (must be an integer)
- System returns an integer between 0 and p-1, where p is the size of the communicator

## Shut Down MPI

```
MPI_Finalize ();
```

- Call after the last call to all other MPI library calls
- Allows system to free up resources

## The program

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i, id, p;
    void check_circuit (int, int);

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);

    for (i = id; i < 65536; i += p) check_circuit (id, i);

    printf ("Process %d is done\n", id);
    fflush (stdout);
    MPI_Finalize();
    return 0;
}
```

## Execution (1 processor)

```
0) 1010111110011001
0) 0110111110011001
0) 1110111110011001
0) 101011111011001
0) 011011111011001
0) 111011111011001
0) 101011111011001
0) 011011111011001
0) 111011111011001
Process 0 is done
```

## Execution (3 processors)

```
0) 0110111110011001
0) 1110111111011001
2) 1010111110011001
1) 1110111110011001
1) 1010111111011001
1) 0110111110111001
0) 1010111110111001
2) 0110111111011001
2) 1110111110111001
Process 1 is done
Process 2 is done
Process 0 is done
```

## Observations

- Output order may be non-deterministic
  - Within a single process order is deterministic
  - When multiple processes call I/O (access to a shared resource on the system) there is no guarantee that the “order” is preserved.

## Extensions

- Modify the program to count the total number of solutions
  - First true communication activity
  - Modify `check_circuit` to return 1 or 0
  - Each process keeps a count
  - Perform a sum reduction after the `for` loop

## Code modifications

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i, id, p, count, total;
    int check_circuit (int, int);

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    count = 0;
    for (i = id; i < 65536; i += p) count += check_circuit (id, i);
    /* Now do an MPI_Reduce */
    printf ("Process %d is done\n", id);
    fflush (stdout);
    MPI_Finalize();
    return 0;
}
```

## MPI Reduce

```
int MPI_Reduce(  
    void      *operand, /* address of start of the operand buffer */  
    void      *result,  /* address of start of the result buffer */  
    int       count,    /* size of the operand buffer */  
    MPI_Datatype type,  /* type of the elements */  
    MPI_Op    operator, /* operator */  
    int       root,     /* id of the root processor */  
    MPI_Comm  comm,    /* communicator */  
)
```

## Final Code

```
#include <mpi.h>  
#include <stdio.h>  
  
/* all the declarations and code unchanged until the for loop */  
  
    for (i = id; i < 65536; i += p) count += check_circuit (id, i);  
    /* Now do an MPI_Reduce */  
  
        MPI_Reduce (&count,  
                    &total,  
                    1,  
                    MPI_INT,  
                    MPI_SUM,  
                    0,  
                    MPI_COMM_WORKD);  
    if (id==0) printf ( "There are %d solutions" , total);  
}
```

## Summary (1)

- MPI is the most widely used parallel programming API
- Portable across many different platforms
- Provides high performance
- Still the “assembly language” of parallel programming

## Summary (2) MPI Functions

- `MPI_Init`
- `MPI_Comm_rank`
- `MPI_Comm_size`
- `MPI_Reduce`
- `MPI_Finalize`
- Others (in lab): `MPI_Barrier`, `MPI_Wtime`, `MPI_Wtick`