

Parallel Programming in C with MPI and OpenMP

Michael J. Quinn



Chapter 4

Message-Passing Programming

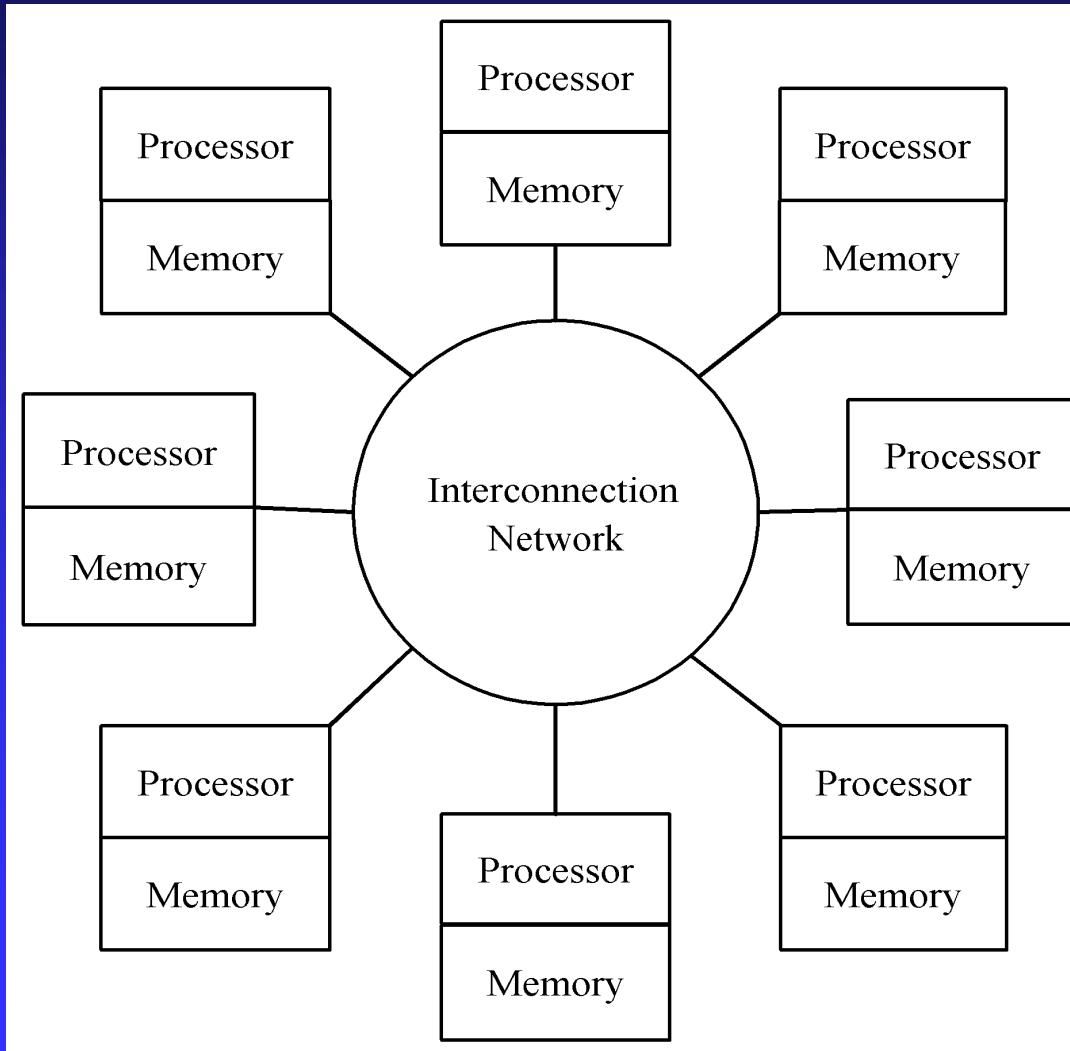
Learning Objectives

- Understanding how MPI programs execute
- Familiarity with fundamental MPI functions

Outline

- Message-passing model
- Message Passing Interface (MPI)
- Coding MPI programs
- Compiling MPI programs
- Running MPI programs
- Benchmarking MPI programs

Message-passing Model



Task/Channel vs. Message-passing

Task/Channel	Message-passing
Task	Process
Explicit channels	Any-to-any communication

Processes

- Number is specified at start-up time
- Remains constant throughout execution of program
- All execute same program
- Each has unique ID number
- Alternately performs computations and communicates

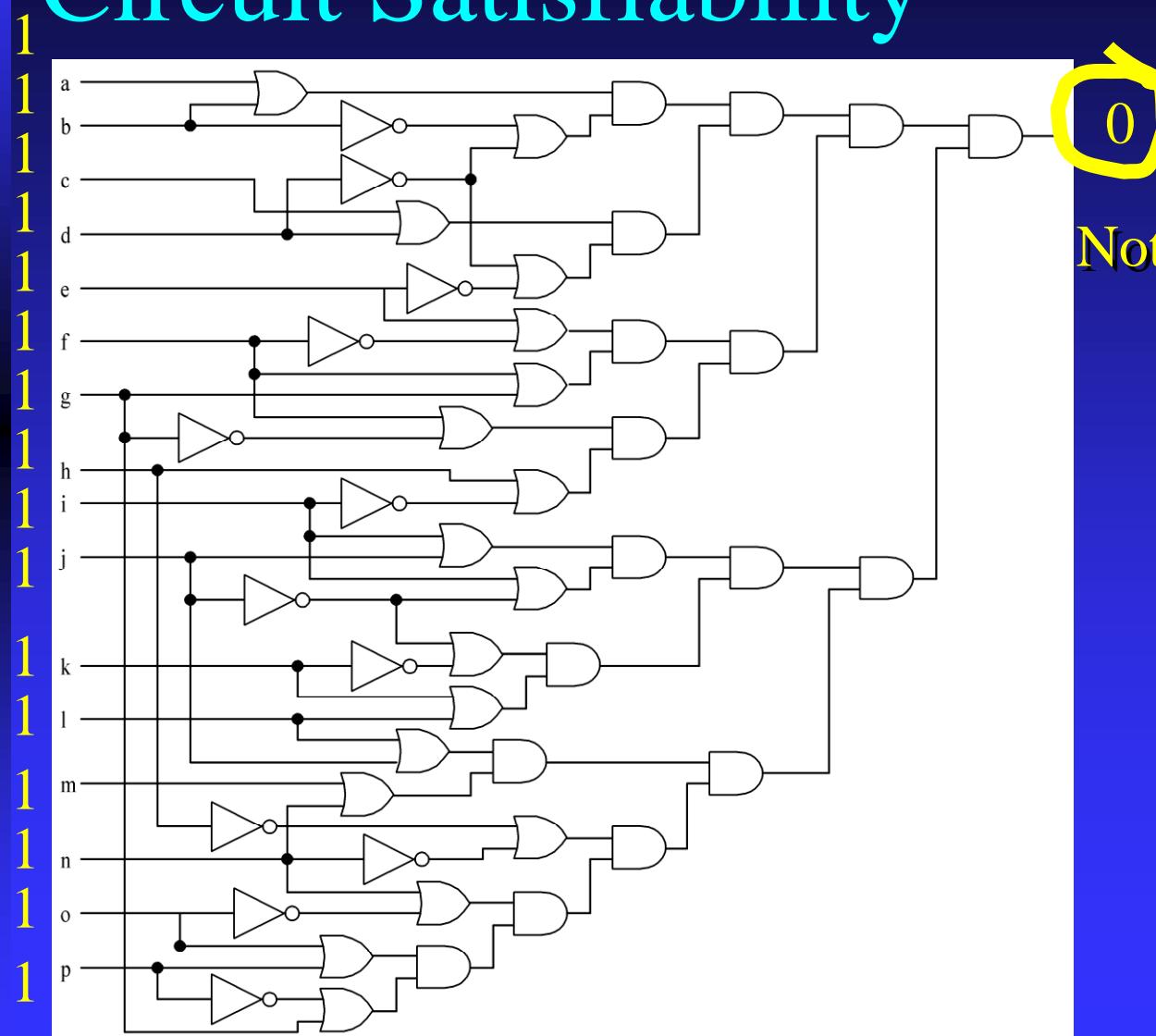
Advantages of Message-passing Model

- Gives programmer ability to manage the memory hierarchy
- Portability to many architectures
- Easier to create a deterministic program
- Simplifies debugging

The Message Passing Interface

- Late 1980s: vendors had unique libraries
- 1989: Parallel Virtual Machine (PVM) developed at Oak Ridge National Lab
- 1992: Work on MPI standard begun
- 1994: Version 1.0 of MPI standard
- 1997: Version 2.0 of MPI standard
- Today: MPI is dominant message passing library standard

Circuit Satisfiability

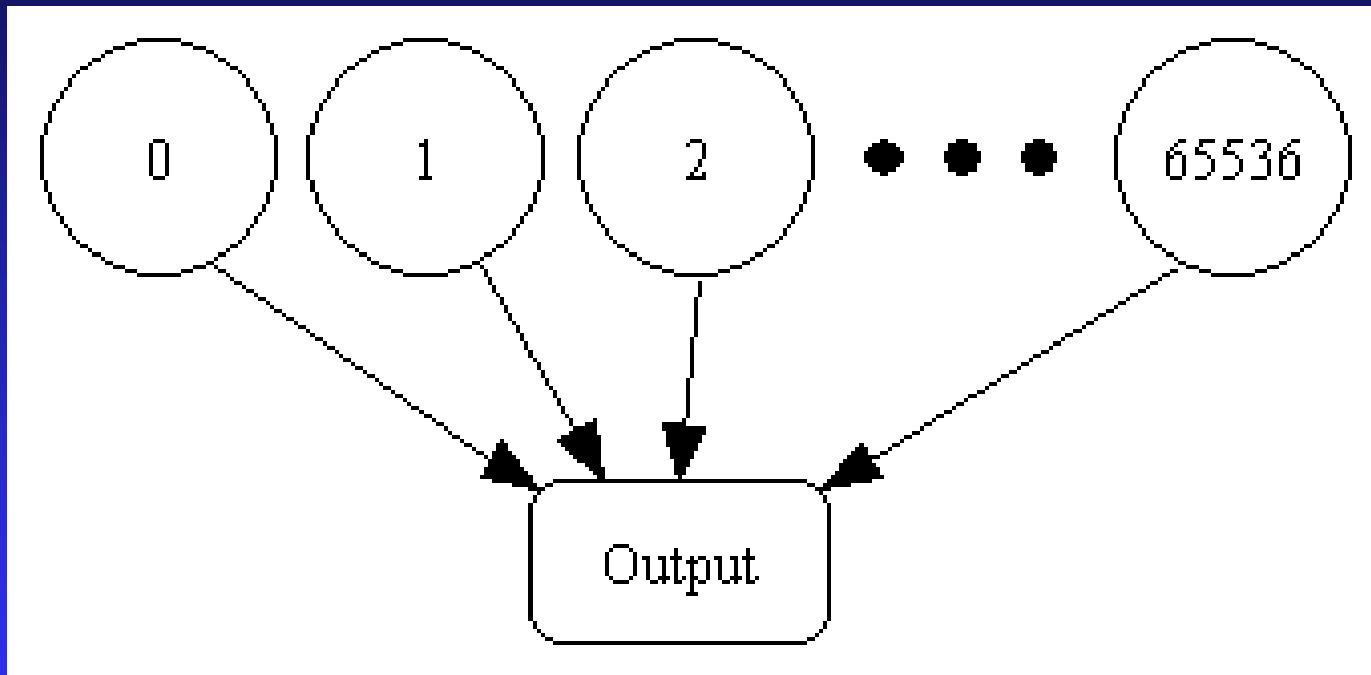


Not satisfied

Solution Method

- Circuit satisfiability is NP-complete
- No known algorithms to solve in polynomial time
- We seek all solutions
- We find through exhaustive search
- 16 inputs \Rightarrow 65,536 combinations to test

Partitioning: Functional Decomposition



- **Embarassingly parallel:** No channels between tasks

Agglomeration and Mapping

- Properties of parallel algorithm
 - ◆ Fixed number of tasks
 - ◆ No communications between tasks
 - ◆ Time needed per task is variable
- Consult mapping strategy decision tree
 - ◆ Map tasks to processors in a cyclic fashion

Cyclic (interleaved) Allocation

- Assume p processes
- Each process gets every p^{th} piece of work
- Example: 5 processes and 12 pieces of work
 - ◆ P_0 : 0, 5, 10
 - ◆ P_1 : 1, 6, 11
 - ◆ P_2 : 2, 7
 - ◆ P_3 : 3, 8
 - ◆ P_4 : 4, 9

Pop Quiz

- Assume n pieces of work, p processes, and cyclic allocation
- What is the most pieces of work any process has?
- What is the least pieces of work any process has?
- How many processes have the most pieces of work?

Summary of Program Design

- Program will consider all 65,536 combinations of 16 boolean inputs
- Combinations allocated in cyclic fashion to processes
- Each process examines each of its combinations
- If it finds a satisfiable combination, it will print it

Include Files

```
#include <mpi.h>
```

- MPI header file

```
#include <stdio.h>
```

- Standard I/O header file

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**: they are needed to initialize MPI
- One copy of every variable for each process running this program

Initialize MPI

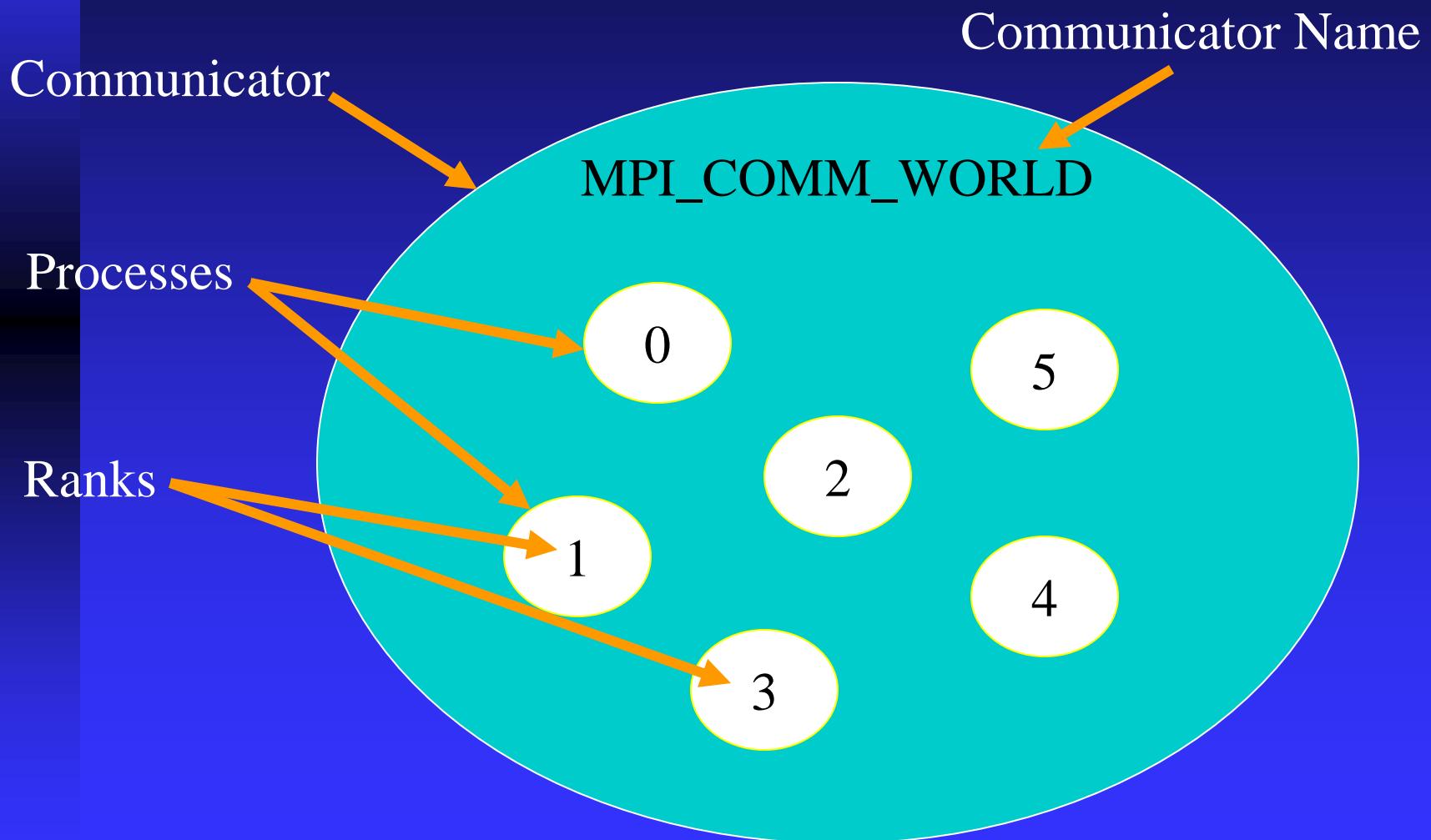
```
MPI_Init (&argc, &argv) ;
```

- First MPI function called by each process
- Not necessarily first executable statement
- Allows system to do any necessary setup

Communicators

- Communicator: opaque object that provides message-passing environment for processes
- MPI_COMM_WORLD
 - ◆ Default communicator
 - ◆ Includes all processes
- Possible to create new communicators
 - ◆ Will do this in Chapters 8 and 9

Communicator



Determine Number of Processes

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

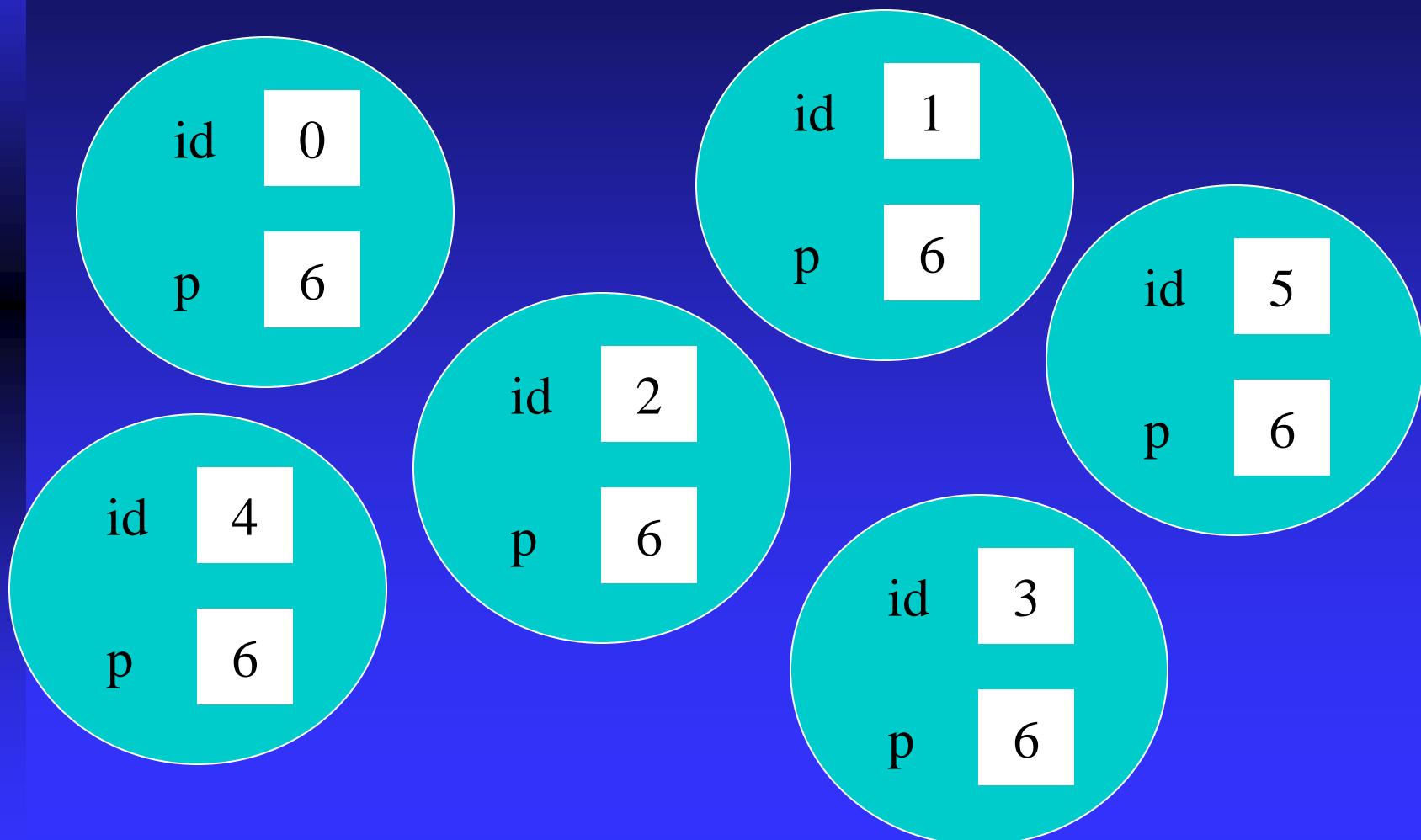
- First argument is communicator
- Number of processes returned through second argument

Determine Process Rank

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

- First argument is communicator
- Process rank (in range 0, 1, ..., $p-1$) returned through second argument

Replication of Automatic Variables



What about External Variables?

```
int total;
```

```
int main (int argc, char *argv[]) {  
    int i;  
    int id;  
    int p;  
    ...
```

- Where is variable **total** stored?

Cyclic Allocation of Work

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

- Parallelism is outside function
check_circuit
- It can be an ordinary, sequential function

Shutting Down MPI

```
MPI_Finalize();
```

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

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

int main (int argc, char *argv[]) {
    int i;
    int id;
    int 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;
}
```

Put fflush() after every printf()

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

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\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);
    }
}
```

Compiling MPI Programs

```
mpicc -O -o foo foo.c
```

- **mpicc**: script to compile and link C+MPI programs
- Flags: same meaning as C compiler
 - ◆ **-O** — optimize
 - ◆ **-o <file>**— where to put executable

Running MPI Programs

- **mpirun -np <p> <exec> <arg1> ...**
 - ◆ **-np <p>** — number of processes
 - ◆ **<exec>** — executable
 - ◆ **<arg1> ...** — command-line arguments

Specifying Host Processors

- File **.mpi-machines** in home directory lists host processors in order of their use
- Example **.mpi_machines** file contents
 - band01.cs.ppu.edu**
 - band02.cs.ppu.edu**
 - band03.cs.ppu.edu**
 - band04.cs.ppu.edu**

Enabling Remote Logins

- MPI needs to be able to initiate processes on other processors without supplying a password
- Each processor in group must list all other processors in its **.rhosts** file; e.g.,

band01.cs.ppu.edu student

band02.cs.ppu.edu student

band03.cs.ppu.edu student

band04.cs.ppu.edu student

Execution on 1 CPU

```
% mpirun -np 1 sat
0) 101011110011001
0) 011011110011001
0) 111011110011001
0) 101011111011001
0) 011011111011001
0) 111011111011001
0) 1010111110111001
0) 0110111110111001
0) 1110111110111001
Process 0 is done
```

Execution on 2 CPUs

```
% mpirun -np 2 sat
0) 0110111110011001
0) 0110111111011001
0) 0110111110111001
1) 1010111110011001
1) 1110111110011001
1) 101011111011001
1) 1110111111011001
1) 1010111110111001
1) 1110111111011001
Process 0 is done
Process 1 is done
```

Execution on 3 CPUs

```
% mpirun -np 3 sat
0) 0110111110011001
0) 1110111111011001
2) 1010111110011001
1) 1110111110011001
1) 1010111111011001
1) 0110111110111001
0) 1010111110111001
2) 0110111111011001
2) 11101111110111001
Process 1 is done
Process 2 is done
Process 0 is done
```

Deciphering Output

- Output order only partially reflects order of output events inside parallel computer
- If process A prints two messages, first message will appear before second
- If process A calls **printf** before process B, there is no guarantee process A's message will appear before process B's message

Enhancing the Program

- We want to find total number of solutions
- Incorporate sum-reduction into program
- Reduction is a **collective communication**

Modifications

- Modify function **check_circuit**
 - ◆ Return 1 if circuit satisfiable with input combination
 - ◆ Return 0 otherwise
- Each process keeps local count of satisfiable circuits it has found
- Perform reduction after **for** loop

New Declarations and Code

```
int count; /* Local sum */  
int global_count; /* Global sum */  
int check_circuit (int, int);  
  
count = 0;  
for (i = id; i < 65536; i += p)  
    count += check_circuit (id, i);
```

Prototype of **MPI_Reduce()**

```
int MPI_Reduce (
    void          *operand,
                  /* addr of 1st reduction element */
    void          *result,
                  /* addr of 1st reduction result */
    int           count,
                  /* reductions to perform */
    MPI_Datatype type,
                  /* type of elements */
    MPI_Op        operator,
                  /* reduction operator */
    int           root,
                  /* process getting result(s) */
    MPI_Comm      comm
                  /* communicator */
)
```

MPI_Datatype Options

- **MPI_CHAR**
- **MPI_DOUBLE**
- **MPI_FLOAT**
- **MPI_INT**
- **MPI_LONG**
- **MPI_LONG_DOUBLE**
- **MPI_SHORT**
- **MPI_UNSIGNED_CHAR**
- **MPI_UNSIGNED**
- **MPI_UNSIGNED_LONG**
- **MPI_UNSIGNED_SHORT**

MPI_Op Options

- **MPI_BAND**
- **MPI_BOR**
- **MPI_BXOR**
- **MPI_LAND**
- **MPI_LOR**
- **MPI_LXOR**
- **MPI_MAX**
- **MPI_MAXLOC**
- **MPI_MIN**
- **MPI_MINLOC**
- **MPI_PROD**
- **MPI_SUM**

Our Call to **MPI_Reduce()**

```
MPI_Reduce (&count,  
            &global_count,  
            1,  
            MPI_INT,  
            MPI_SUM,  
            0,
```

Only process 0

will get the result

```
                MPI_COMM_WORLD);
```

```
if (!id) printf ("There are %d different solutions\n",  
global_count);
```

Execution of Second Program

```
% mpirun -np 3 seq2
0) 0110111110011001
0) 1110111111011001
1) 1110111110011001
1) 1010111111011001
2) 1010111110011001
2) 0110111111011001
2) 11101111110111001
1) 01101111110111001
0) 10101111110111001
Process 1 is done
Process 2 is done
Process 0 is done
There are 9 different solutions
```

Benchmarking the Program

- **MPI_Barrier** — barrier synchronization
- **MPI_Wtick** — timer resolution
- **MPI_Wtime** — current time

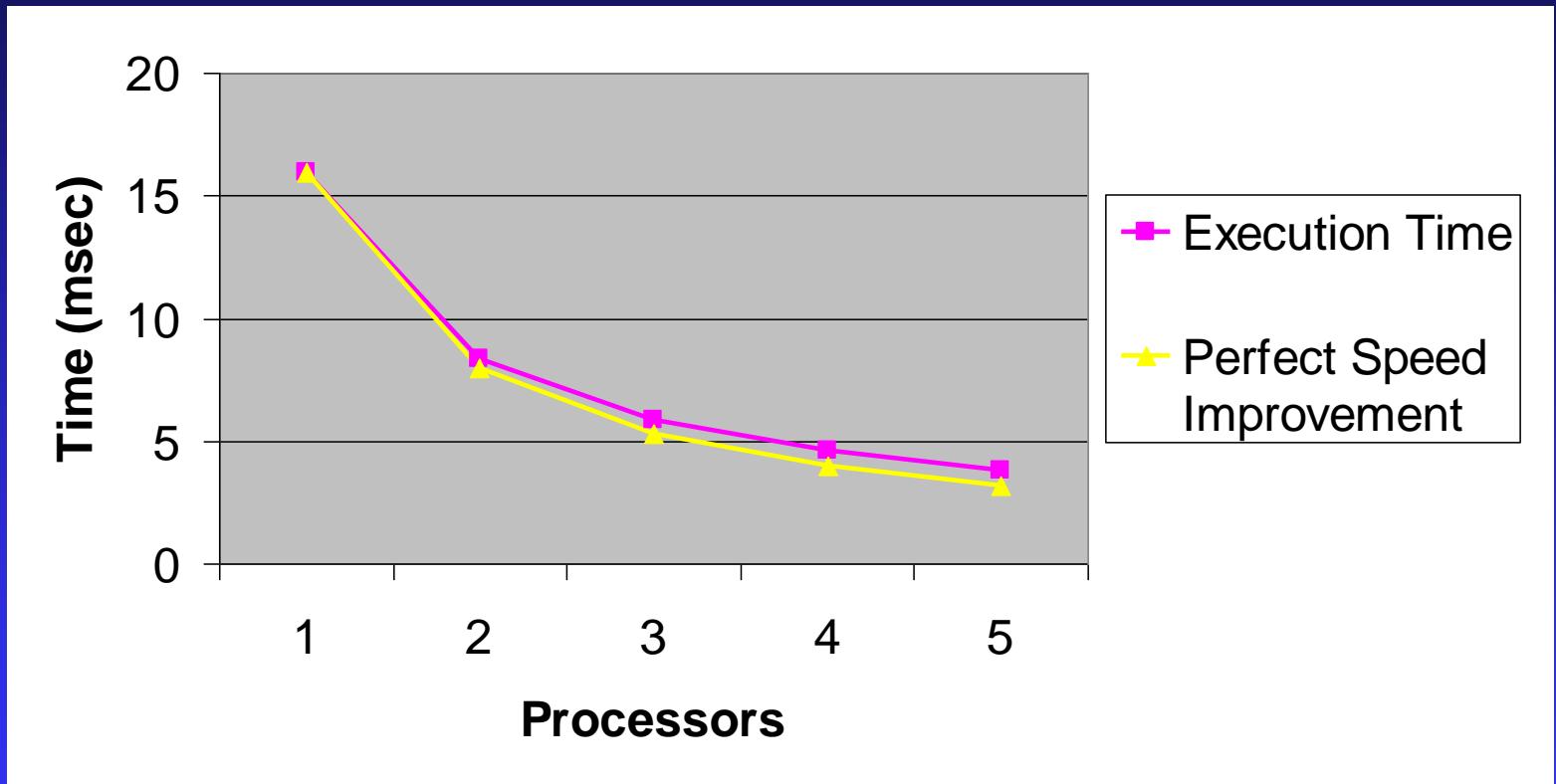
Benchmarking Code

```
double elapsed_time;  
...  
MPI_Init (&argc, &argv);  
MPI_Barrier (MPI_COMM_WORLD);  
elapsed_time = - MPI_Wtime();  
...  
MPI_Reduce (...);  
elapsed_time += MPI_Wtime();
```

Benchmarking Results

Processors	Time (sec)
1	15.93
2	8.38
3	5.86
4	4.60
5	3.77

Benchmarking Results



Summary (1/2)

- Message-passing programming follows naturally from task/channel model
- Portability of message-passing programs
- MPI most widely adopted standard

Summary (2/2)

- MPI functions introduced

- ◆ **MPI_Init**
- ◆ **MPI_Comm_rank**
- ◆ **MPI_Comm_size**
- ◆ **MPI_Reduce**
- ◆ **MPI_Finalize**
- ◆ **MPI_Barrier**
- ◆ **MPI_Wtime**
- ◆ **MPI_Wtick**