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

<table>
<thead>
<tr>
<th>Task/Channel</th>
<th>Message-passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>Process</td>
</tr>
<tr>
<td>Explicit channels</td>
<td>Any-to-any communication</td>
</tr>
</tbody>
</table>
Processes

- Number is specified at start-up time
- Remains constant throughout execution of program
- All execute same program (SPMD)
- 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
  - But deadlock is possible
- 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
  - what does that mean?
- We seek all solutions through exhaustive search
- 16 inputs $\Rightarrow$ 65,536 combinations to test
  - $p$ processes ($0 .. p-1$):
    - $i$ 16 bit number ($0 .. 65535$)
    - process $j$ checks $i=j+kp$, $k = 0 .. 65535/p$
Agglomeration and Mapping

- Properties of parallel SAT algorithm
  - Fixed number of tasks
  - Minimal communications between tasks
    - just to produce the results
  - Time needed per task is variable
  - 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

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

- **MPI header file**

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

- **Standard I/O header file**
Local Variables

```c
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 name space that provides message-passing environment for processes

- MPI_COMM_WORLD
  - Default communicator
  - Includes all processes

- Possible to create new communicators
Determine Number of Processes

\texttt{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
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
```c
#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);

        && (v[14] || v[15])) {
        printf ("%d %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);
    }
}
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

- A file lists host processors in order of their use
- run as follows:
  mpirun –np 2 --hostfile hosts sat

(see PA4 JacMPI )
Execution on 1 CPU

```bash
% mpirun -np 1 sat
0) 10101111110011001
0) 01101111110011001
0) 11101111110011001
0) 1010111111011001
0) 0110111111011001
0) 1110111111011001
0) 1010111110111001
0) 0110111110111001
0) 1110111110111001
Process 0 is done
```
Execution on 2 CPUs

```plaintext
% mpirun -np 2 sat
0) 0110111110011001
0) 0110111111011001
0) 0110111110111001
1) 1010111110011001
1) 1110111110011001
1) 1010111111011001
1) 1110111111011001
1) 1010111110111001
1) 1110111110111001
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) 0110111110111001
2) 1110111110111001
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.
- My style:
  - have a verbose parameter: if it is there, it indicates which process “speaks”
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

```c
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()`

```c
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()`

```c
MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Only process 0 will get the result

```c
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) 1110111110111001
1) 0110111110111001
0) 1010111110111001
0) 1010111110111001

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

- We use our standard timer.c and timer.h
Benchmarking Results

<table>
<thead>
<tr>
<th>Processors</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.93</td>
</tr>
<tr>
<td>2</td>
<td>8.38</td>
</tr>
<tr>
<td>3</td>
<td>5.86</td>
</tr>
<tr>
<td>4</td>
<td>4.60</td>
</tr>
<tr>
<td>5</td>
<td>3.77</td>
</tr>
</tbody>
</table>
Benchmarking Results

![Graph showing benchmarking results](Image)
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`
More MPI functions

Quinn Chapters 5 and 6
Function MPI_Bcast

```c
int MPI_Bcast ( 
    void *buffer,    /* Addr of 1st element */
    int count,       /* # elements to broadcast */
    MPI_Datatype datatype, /* Type of elements */
    int root,        /* ID of root process */
    MPI_Comm comm)  /* Communicator */

MPI_Bcast (&k, 1, MPI_INT, 0, MPI_COMM_WORLD);
```
Function MPI_Send

```c
int MPI_Send (  
    void         *message,  
    int           count,  
    MPI_Datatype  datatype,  
    int           dest,  
    int           tag,  
    MPI_Comm      comm
)
```
Function MPI_Recv

```c
int MPI_Recv (  
    void         *message,  
    int           count,  
    MPI_Datatype  datatype,  
    int           source,  
    int           tag,  
    MPI_Comm      comm,  
    MPI_Status   *status  
)  
```
Coding Send/Receive

... if (ID == j) {
    ...
    Receive from i
    ...
} 

... if (ID == i) {
    ...
    Send to j
    ...
} 

...
Inside MPI_Send and MPI_Recv
Return from MPI_Send

- Function blocks until message buffer free
- Message buffer is free when
  - Message copied to system buffer, or
  - Message transmitted
- Typical scenario
  - Message copied to system buffer
  - Transmission overlaps computation
Return from MPI_Recv

- Function blocks until message in buffer
- If message never arrives, function never returns
Deadlock

- Deadlock: process waiting for a condition that will never become true
- Easy to write send/receive code that deadlocks
  - Two processes: both receive before send
  - Send tag doesn’t match receive tag
  - Process sends message to wrong destination process