Chapter 4
Message-Passing Programming

Message-passing Model
Characteristics of Processes

• Number is specified at start-up time
• Remains constant throughout the execution of program
• All execute same program
• Each has unique ID number
• Alternately performs computations and communicates
• Passes messages both to communicate and to synchronize with each other.

Features of Message-passing Model

• Runs well on a variety of MIMD architectures.
  – Natural fit for multicomputers
• Execute on multiprocessors by using shared variables as message buffers
  – Model’s distinction between faster, directly accessible local memory and slower, indirectly accessible remote memory encourages designing algorithms that maximize local computation and minimize communications
• Simplifies debugging
  – Easier than debugging shared-variable programs
Message Passing Interface History

- Late 1980s: vendors had unique libraries
  - Usually FORTRAN or C augmented with functions calls that supported message-passing
- 1989: Parallel Virtual Machine (PVM) developed at Oak Ridge National Lab
  - Supported execution of parallel programs across a heterogeneous group of parallel and serial computers
- 1992: Work on MPI standard began
  - Chose best features of earlier message passing languages
  - Not for heterogeneous setting – i.e., homogeneous
- Today: MPI is dominant message passing library standard

What We Will Assume

- The programming paradigm typically used with MPI is called a SPMD paradigm (single program multiple data)
- Consequently, the same program runs on each processor
- The effect of running different programs is achieved by branches within the source code where different processors execute different branches
Circuit Satisfiability Problem

Given a circuit containing AND, OR, and NOT gates, find if there are any combinations of input 0/1 values for which the circuit output is the value 1

Note: The input consists of variables a, b, ..., p
Solution Method

- Circuit satisfiability is NP-complete
  - What combinations of input values will the circuit output the value 1
- We seek all solutions
  - Not a “Yes/No” answer about solution existing
- We find solutions using exhaustive search
  - 16 inputs \( \Rightarrow 2^{16} = 65,536 \) combinations to test
- Functional decomposition natural here

Embarrassingly Parallel

- The problem solution falls easily into the definition of tasks that do not need to interact with each other, then the problem is said to be embarrassingly parallel
- H.J. Siegel calls this situation instead pleasingly parallel and many professionals use this term
Partitioning: Functional Decomposition

- **Embarrassingly (or pleasingly) parallel**

Agglomeration and Mapping

- Properties of parallel algorithm
  - Fixed number of tasks
  - No communications between tasks
  - Time needed per task is variable
    - Bit sequences for most tasks do not satisfy circuit
    - Some bit sequences are quickly seen unsatisfiable
    - Other bit sequences may take more time
- 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
  - i.e., each piece of work, \( I \), is assigned to process \( k \)
    where \( k = i \mod 5 \)

Questions to Consider

- Assume \( n \) pieces of work, \( p \) processes, and cyclic allocation
- What is the maximum pieces of work any process has?
- What is the minimum pieces of work any process has?
- How many processes have the most pieces of work?
Summary of Program Design

• Program considers 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 prints this combination

MPI Program for Circuit Satisfiability

• Each active MPI process executes its own copy of the program

• Each process has its own copy of all the variables declared in the program, including:
  – External variables declared outside of any function
  – Automatic variables declared inside a function
C Code Include Files

#include <mpi.h> /* MPI header file */
#include <stdio.h> /* Standard C I/O header file */

• These appear at the beginning of the program file.

• The file name will have a .c as these are C programs, augmented with the MPI library.

Header for C Function Main
(Local Variables)

int main (int argc, char *argv[]) {
  int i; /* loop index */
  int id; /* Process ID number */
  int p; /* Number of processes */
  void check_circuit (int, int);

  Include argc and argv: they are needed to initialize MPI
  The i, id, and p are local (or automatic) variables.
  One copy of every variable is needed for each process running this program
  If there are p processes, then the ID numbers start at 0 and end at p -1.
Replication of Automatic Variables
(Shown for id and p only)

Initialize MPI

MPI_Init (&argc, &argv);

• First MPI function called by each process
• Not necessarily first executable statement
• In fact, call need not be located in main
• But, it must be called before any other MPI function is invoked
• Allows system to do any necessary setup to handle calls to MPI library
MPI Identifiers

• All MPI identifiers (including function identifiers) begin with the prefix “MPI_”
• The next character is a capital letter followed by a series of lowercase letters and underscores.
• Example: MPI_Init
• All MPI constants are strings of capital letters and underscores beginning with MPI_
• Recall C is case-sensitive as it was developed in a UNIX environment.

Communicators

• When MPI is initialized, every active process becomes a member of a communicator called MPI_COMM_WORLD.
• Communicator: Opaque object that provides the message-passing environment for processes
• MPI_COMM_WORLD
  – This is the default communicator
  – It includes all processes automatically
  – For most programs, this is sufficient
• It is possible to create new communicators
  – These are needed if you need to partition the processes into independent communicating groups
Communicators (cont.)

• Processes within a communicator are ordered
• The rank of a process is its position in the overall order
• In a communicator with $p$ processes, each process has a unique rank, which we often think of as an ID number, between 0 and $p-1$
• A process may use its rank to determine the portion of a computation or portion of a dataset that it is responsible for
Determine Process Rank

MPI_Comm_rank (MPI_COMM_WORLD, &id);

- A process can call this function to determine its rank with a communicator
- The first argument is the communicator name
- The process rank (in range 0, 1, ..., \(p-1\)) is returned through second argument

Determine Number of Processes

MPI_Comm_size (MPI_COMM_WORLD, &p);

- A process can call this MPI function
- First argument is the communicator name
- This call determines the number of processes
- The number of processes is returned through the second argument
What about External Variables or Global Variables?

```c
int total;

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

  Try to avoid them
  They can cause major debugging problems.
  However, sometimes they are needed
```

Cyclic Allocation of Work

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

- Now that the MPI process knows its rank and the total number of processes, it may check its share of the 65,536 possible inputs to the circuit
- For example, if there are 5 processes, process id = 3
  checks i = id = 3
    i += 5 = 8
    i += 5 = 13 etc.
- Parallelism is in the outside function check_circuit
- It can be an ordinary, sequential function
After the Loop Completes

printf ("Process %d is done\n", id);
fflush (stdout);
• After the process completes the loop, its work is finished and it prints a message that it is done
• It then flushes the output buffer to ensure the eventual appearance of the message on standard output even if the parallel program crashes
• Put an fflush command after each printf command
• The printf is the standard output command for C. The %d says integer data is to be output and the data appears after the comma – i.e. insert the id number in its place in the text

Shutting Down MPI

MPI_Finalize();
return 0;
• Call after all other MPI library calls
• Allows system to free up MPI resources
• Return code:
  – 0 means the code ran to completion
  – 1 is used to signal an error has occurred
#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;
}

Enhancing the Program

- We want to find the total number of solutions
- A single process can maintain an integer variable that holds the number of solutions it finds, but we want the processors to cooperate to compute the global sum of the values
- Said another way, we want to incorporate a sum-reduction into program. This will require message passing
- Reduction is a collective communication –
  - i.e. a communication operation in which a group of processes works together to distribute or gather together a set of one or more values
Modifications

- Modify function check_circuit
  - Return 1 if the circuit is satisfiable with the input combination
  - Return 0 otherwise
- Each process keeps local count of satisfiable circuits it has found
- We perform reduction after the ‘for’ loop

Modifications

- In function main we need to add two variables:
  - An integer solutions – This keeps track of solutions for this process
  - An integer global_solutions – This is used only by process 0 to store the grand total of the count values from the other processes
  - Process 0 is also responsible for printing the total count at the end
  - Remember that each process runs the same program, but if statements and various assignment statements dictate which code a process executes
New Declarations and Code

```c
int solutions; /* Local sum */
int global_solutions; /* Global sum */
int check_circuit (int, int);

solutions = 0;
for (i = id; i < 65536; i += p)
    solutions += check_circuit (id, i);
```

This loop calculates the total number of solutions for each individual process. We now have to collect the individual values with a reduction operation,

The Reduction

- After a process completes its work, it is ready to participate in the reduction operation.
- MPI provides a function, MPI_Reduce, to perform one or more reduction operation on values submitted by all the processes in a communicator.
- The next slide shows the header for this function and the parameters we will use.
- Most of the parameters are self-explanatory.
Header for 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 */
)
```

Our call will be:

```c
MPI_Reduce (&solutions, &global_solutions, 1,
            MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

**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 for Reduce**

- MPI_BAND \( B = \text{bitwise} \)
- MPI_BOR
- MPI_BXOR
- MPI_BAND \( L = \text{logical} \)
- MPI_LAND
- MPI_LOR
- MPI_LXOR
- MPI_MAX
- MPI_MAXLOC \( \text{Max and location of max} \)
- MPI_MIN
- MPI_MINLOC
- MPI_PROD
- MPI_SUM

---

**Our Call to MPI_Reduce()**

```c
MPI_Reduce (&solutions, &global_solutions, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

If count > 1, list elements for reduction are found in contiguous memory.

Only process 0 will get the result.

After this call, process 0 has in `global_solutions` the sum of all of the other processes `solutions`. We then conditionally execute the print statement:

```c
if (id==0) printf("There are %d different solutions\n", global_solutions);
```
Version 2 of Circuit Satisfiability

• The code for main is on page 105 and incorporates all the changes we made plus we make trivial changes for check_circuit to return the values of 1 or 0.

• First, in main, the declaration must show an integer being returned instead of a void function:
  ```c
  int check_circuit(int, int);
  ```
  and in the function we need to return a 1 if a solution is found and a 0 otherwise.

Main Program, Circuit Satisfiability, Version 2

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

int main (int argc, char *argv[]) {
  int count;            /* Solutions found by this proc */
  int global_count;     /* Total number of solutions */
  int i;
  int id;              /* Process rank */
  int p;               /* Number of processes */
  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);
  MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM, 0,
               MPI_COMM_WORLD);
  printf("Process %d is done\n", id);
  fflush (stdout);
  MPI_Finalize();
  if (!id) printf("There are %d different solutions\n", global_count);
  return 0;
}
```
Some Cautions About Thinking “Right” About MPI Programming

- The printf statement must be a conditional because only process 0 has the total sum at the end.
- That variable is undefined for the other processes.
- In fact, even if all of them had a valid value, you don’t want all of them printing the same message over and over for 9 times!

Some Cautions About Thinking “Right” about MPI Programming

- Every process in the communicator must execute the MPI_Reduce.
- Processes enter the reduction by volunteering the value – they cannot be called by process 0.
- If you fail to have all process in a communicator call the MPI_Reduce, the program will hang at the point the function is executed,
Execution of Second Program with 3 Processes

0) 0110111110011001
0) 1110111111011001
1) 1110111110011001
1) 1010111111011001
2) 1010111110011001
2) 0110111111011001
2) 1110111110111001
1) 0110111110111001
0) 1010111110111001

Process 1 is done
Process 2 is done
Process 0 is done
There are 9 different solutions

Compare this with slide 42.
The same solutions are found, but output order is different.

Benchmarking

Measuring the Benefit for Parallel Execution
Benchmarking – What is It?

- **Benchmarking**: Uses a collection of runs to test how efficient various programs (or machines) are.
- Usually some kind of counting function is used to count various operations.
- Complexity analysis provides a means of evaluating how good an algorithm is
  - Focuses on the asymptotic behavior of algorithm as size of data increases.
  - Does not require you to examine a specific implementation.
- Once you decide to use benchmarking, you must first have a program as well as a machine on which you can run.
- There are advantages and disadvantages to both types of analysis.

---

Benchmarking

- Determining the complexity analysis for ASC algorithms is done as with sequential algorithms since all PEs are working in lockstep.
- Thus, as with sequential algorithms, you basically have to look at your loops to judge complexity.
- Recall that ASC has a performance monitor that counts the number of scalar operations performed and the number of parallel operations performed.
- Then, given data about a specific machine, run times can be estimated.
Benchmarking with MPI

• When running on a parallel machine that is not synchronized as a SIMD is, we have more difficulties in seeing the effect of parallelism by looking at the code.

• Of course, we can always, in that situation, use the wall clock provided the machine is not being shared with anyone else – background jobs can completely louse up your perceptions.

• As with the ASC, we want to exclude some things from our timings:

Benchmarking a Program

• We will use several MPI-supplied functions:

• `double MPI_Wtime (void)`
  – current time
  – By placing a pair of calls to this function, one before code we wish to time and one after that code, the difference will give us the execution time.

• `double MPI_Wtick (void)`
  – timer resolution
  – Provides the precision of the result returned by MPI_Wtime.

• `int MPI_Barrier (MPI_Comm comm)`
  – barrier synchronization
Barrier Synchronization

- Usually encounter this term first in operating systems classes.
- A barrier is a point where no process can proceed beyond it until all processes have reached it.
- A barrier ensures that all processes are going into the covered section of code at more or less the same time.
- MPI processes theoretically start executing at the same time, but in reality they don’t.
- That can throw off timings significantly.
- In the second version, the call to reduce requires all processes to participate.
- Processes that execute early may wait around a lot before stragglers catch up. These processes would report significantly higher computation times than the latecomers.

Barrier Synchronization

- In operating systems you learn how barriers can be implemented in either hardware or software.
- In MPI, a function is provided that implements a barrier.
- All processes in the specified communicator wait at the barrier point.
double elapsed_time; /* local in main */
...
MPI_Init (&argc, &argv);
MPI_Barrier (MPI_COMM_WORLD); /* wait */
elapsed_time = - MPI_Wtime();
... /* timing all in here */

MPI_Reduce (...); /* Call to Reduce */
elapsed_time += MPI_Wtime(); /* stop timer */

As we don’t want to count I/O, comment out the printf and fflush