MPI
The Message-Passing Standard

Paul E. Plassmann

Scalable Scientific Computing Laboratory
Computer Science and Engineering Department
The Pennsylvania State University
Outline

- Parallel computing architecture review
- Parallel programming models
- The message-passing standard (MPI)
- Programming with MPI
- Compiling, running, benchmarking MPI programs

References:


Parallel Architecture Overview

- Parallel programming model closely coupled to parallel computing architectures
- Dominant architectures:
  - Multiprocessor or Shared Memory
    - Universal (shared) address space
  - Multicomputer, Distributed Memory, or Message Passing
    - Distributed address space
Centralized Multiprocessor

- Memory accessed across bus
- Universal (shared) address space
Multicomputer (Asymmetrical)

- memory local to each computer
- address space not shared
Parallel Programming Models

- How do we program parallel computers?

- New language?
  - Many have been tried (C*, Linda, …)
  - Learning curve too steep!

- Parallelizing compilers?
  - Write standard C/C++/Fortran + magical parallel compiler = parallel code
  - Decades of development with little progress

- Standard language + parallel extensions
  - Use C/C++/Fortran +
    - OpenMP: standard for shared memory, or
    - MPI: standard for message-passing
Shared Memory vs. Message-Passing

- How well do these programming models work with the corresponding architecture?

<table>
<thead>
<tr>
<th></th>
<th>Shared Memory (OpenMP)</th>
<th>Message-Passing (MPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiprocessor</td>
<td>Excellent</td>
<td>Very Good</td>
</tr>
<tr>
<td>Multicomputer</td>
<td>Terrible</td>
<td>Excellent</td>
</tr>
</tbody>
</table>
The Message-Passing Model

- Data is local to each processor (processor memory is distributed, with local address spaces)
- To communicate local data between processors, messages must be sent between processes over network
Single Program Multiple Data (SPMD)

- Each processor runs the same program
- Data different on processes
  - E.g., on process 0: $x = 2.01$
  - on process 1: $x = 3.14$
- Processes differentiated by process IDs
  - E.g., if we have 4 processes, IDs = 0, 1, 2, 3
- Data communicated between processes by “sends” and “receives”
  - E.g., on process 0: `send (&x,1);` // send the value x to process 1
  - on process 1: `recv (&x,0);` // receive the value x from proc 0
Single Program Multiple Data (SPMD)

• Example: write down which statements are executed on processes 0, 1, and 2

```c
int myID;
double x, y;

myID = getProcessID( );

if(myID == 0 ) {
    x = 2.01;
    send(&x,1);
    recv(&y,1);
}
else if (myID == 1) {
    y = 3.14;
    recv(&x,0);
    send(&y,0);
}
```
Solution to Example

Process 0

```c
myID = getProcessID();
    [myID now equal 0]
x = 2.01;
send(&x,1);
recv(&y,1);
    [y now equal 3.14]
```

Process 1

```c
myID = getProcessID();
    [myID now equal 1]
y = 3.14;
recv(&x,0);
    [x now equal 2.01]
send(&y,0);
```

Process 2

```c
myID = getProcessID();
    [myID now equal 2]
```
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 with other processes
Advantages of Message-passing Model

- Gives programmer ability to manage the memory hierarchy
- Portability of programs across 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
Example: Circuit Satisfiability

$2^{16} = 65,536$ possible inputs

Not satisfied
Solution Method

- Circuit satisfiability is NP-complete
  - No known algorithms to solve in polynomial time
- We seek all solutions
  - Determined through exhaustive search
- 16 binary inputs → 65,536 combinations to test
Partitioning: Functional Decomposition

- **Embarrassingly parallel**: No channels between tasks
Mapping Tasks to Processes

• Properties of parallel algorithm
  – Fixed number of tasks
  – No communications between tasks
  – Time needed per task is variable

• Task mapping strategy
  – Map tasks to processors in a cyclic fashion
  – Should minimize any task time discrepancies
Cyclic (interleaved) Allocation

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

• 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
• We will develop a C + MPI program
Include Files

```c
#include <mpi.h>

• MPI header file

#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
• Should be 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
  - Often used in “divide and conquer” algorithms
Communicators

MPI_COMM_WORLD

Communicator Name

Communicator

Processes

Ranks
int p;
    MPI_Comm_size (MPI_COMM_WORLD, &p);

• First argument is communicator
• Number of processes returned through second argument (note that address of p is passed!)
Determine Process Rank

```c
int id;
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?

```c
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;
}
/* 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

- **Batch scheduler**
  - May also use PBS
  - `qsub pbs.sh`
% mpirun -np 1 sat
0) 1010111110011001
0) 0110111110011001
0) 1110111110011001
0) 1010111111011001
0) 0110111111011001
0) 1110111111011001
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) 1010111111011001
1) 1110111111011001
1) 1010111110111001
1) 1110111110111001
Process 0 is done
Process 1 is done
% mpirun -np 3 sat
0) 0110111110011001
0) 1110111111011001
2) 1010111110011001
1) 1110111110011001
1) 1010111111011001
1) 0110111111011001
1) 0110111110111001
0) 1010111110111001
0) 1010111111011001
2) 0110111111011001
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.
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
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 \texttt{MPI\_Reduce()}

\begin{verbatim}
MPI\_Reduce (&count,
             &global\_count,
             1,
             MPI\_INT,
             MPI\_SUM,
             0,
             MPI\_COMM\_WORLD);
\end{verbatim}

Only process 0 will get the result

\begin{verbatim}
if (!id) printf ("There are %d different solutions\n", global\_count);
\end{verbatim}
% mpirun -np 3 seq2
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
Benchmarking the Program

- **MPI_Barrier** — barrier synchronization
- **MPI_Wtick** — timer resolution
- **MPI_Wtime** — current time
double elapsed_time;
...
MPI_Init (&argc, &argv);
MPI_Barrier (MPI_COMM_WORLD);
elapsed_time = - MPI_Wtime();
...
MPIReduce (...);
elapsed_time += MPI_Wtime();
## 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 with Processors on the x-axis and Time (msec) on the y-axis. The graph compares Execution Time and Perfect Speed Improvement.]
Summary (1/2)

• Message-passing programming closely associated with the multicomputer architecture
• MPI ensures portability of message-passing programs
• MPI a widely adopted standard
• Did not get to MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv, etc.
Summary (2/2)

- MPI functions introduced
  - MPI_Init
  - MPI_Comm_rank
  - MPI_Comm_size
  - MPI_Reduce
  - MPI_Finalize
  - MPI_Barrier
  - MPI_Wtime
  - MPI_Wtick