CZ4102 – High Performance Computing

Lectures 8 and 9:

Programming Using the Message Passing Paradigm

- Dr Tay Seng Chuan

Reference:

(i) "Introduction to Parallel Computing" – Chapter 6.

(ii) "Parallel Programming in C with MPI and OpenMP"
– Michael J. Quinn, Mc Graw Hill, ISBN 007-123265-6, Chapter 4.

MPI implementation in CZ lab: http://www-unix.mcs.anl.gov/mpi/mpich/
MPI Document: http://www.mpi-forum.org/docs/docs.html

1

Topic Overview

- Principles of Message-Passing Programming
- The Building Blocks: Send and Receive Operations
- MPI: the Message Passing Interface
- Overlapping Communication with Computation
- MPI Program Examples

2

Principles of Message-Passing Programming

- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- Each data element belong to one of the partitions of the space. Therefore, data must be explicitly partitioned and placed in different processors.
- All interactions (read-only or read/write) require cooperation of two processes - the process that has the data and the process that wants to access the data.
- These underlying communication costs are explicit to the programmer as the instructions on message passing need to be programmed explicitly.

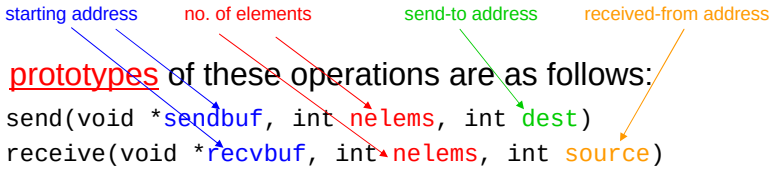
3

Principles of Message-Passing Programming

- Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms.
- In the asynchronous paradigm, all concurrent tasks execute asynchronously.
- In the loosely synchronous model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions called **barrier synchronization**, tasks are executed without any synchronization.
- Message-passing programs are written using the single program multiple data (SPMD) model, or MIMD model.
- MPI program is SPMD.
- Although MPI is based on SPMD model, it is able to emulate the full capability of the MIMD model. **How?**

4

The Building Blocks: Send and Receive Operations



- The prototypes of these operations are as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

P0

```
a = 100;
send(&a, 1, 1);
a = 0;
```

P1

```
receive(&a, 1, 0)
printf("%d\n", a);
```

- The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0. We need a protocol to ensure the correctness of the semantic.

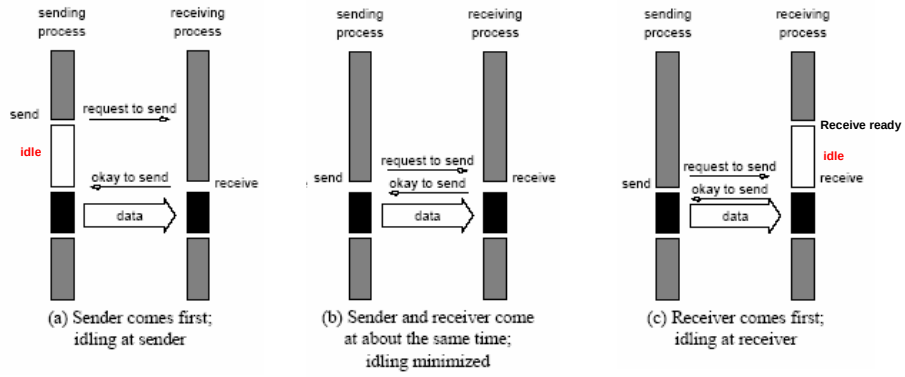
5

Non-Buffered Blocking Message Passing Operations

- A simple method for ensuring send/receive semantics is for the send operation to return only when it is safe to do so.
- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process. This is to ensure the correctness of the semantic.
- Idling and deadlocks are major issues with non-buffered blocking sends.
- In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The data is copied at a buffer at the receiving end as well.
- Buffering alleviates idling time (or idling overhead) at the expense of copying overheads.

6

Non-Buffered Blocking Message Passing Operations



Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

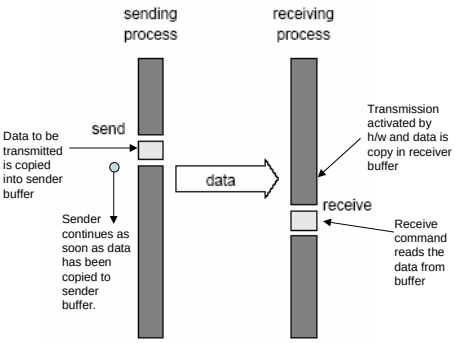
7

Buffered Blocking Message Passing Operations

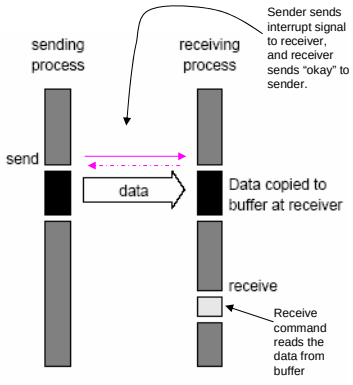
- A simple solution to the idling (and deadlocking which is to be discussed later) problem outlined above is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data must be buffered at the receiving end as well.
- Buffering trades off idling overhead for buffer copying overhead.

8

Buffered Blocking Message Passing Operations



(a) in the presence of communication hardware with buffers at send and receive ends



(b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

9

Buffered Blocking Message Passing Operations

Bounded buffer sizes can have significant impact on performance.

P0

```
for (i = 0; i < 1000; i++)
{
   produce_data(&a);
   send(&a, 1, 1);
}
```

P1

```
for (i = 0; i < 1000; i++)
{
   receive(&a, 1, 0);
   consume_data(&a);
}
```

What if the consumer is much slower than producer?

Answer:

10

Buffered Blocking Message Passing Operations

Deadlocks are still possible with buffering since receive operations block.

P0

```
receive(&a, 1, 1);
send(&b, 1, 1);
```

P1

```
receive(&a, 1, 0);
send(&b, 1, 0);
```

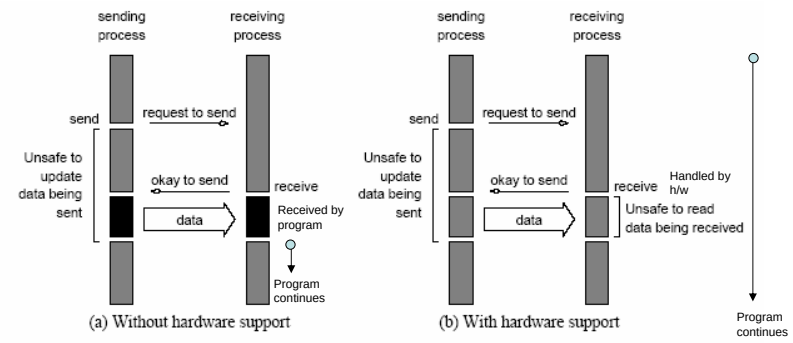
11

Non-Blocking Message Passing Operations

- The programmer must ensure semantics of the blocking send and receive operators.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.
- Non-blocking operations are generally accompanied by a check-status operation.
- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.
- Message passing libraries typically provide both blocking and non-blocking primitives.

12

Non-Blocking Non-Buffered Message Passing Operations



(a) In absence of communication hardware

(b) in presence of communication hardware

13

Send and Receive Protocols

	Blocking Operations	Non-Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non-Buffered	Sending process blocks until matching receive operation has been encountered	
	Send and Receive semantics assured by corresponding operation	Programmer must explicitly ensure semantics by polling to verify completion

Space of possible protocols for send and receive operations.

14

MPI: the Message Passing Interface

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.
- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.
- It is possible to write fully-functional message-passing programs by using only the six routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

The minimal set of MPI routines.

15

Starting and Terminating the MPI Library

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are:

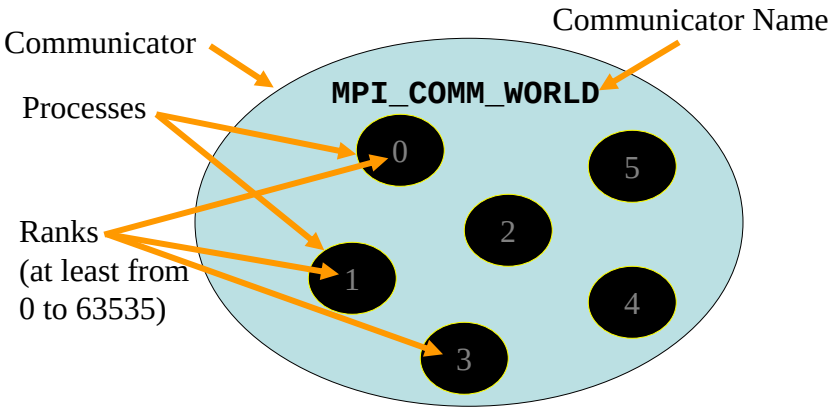
```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

- `MPI_Init` also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by "`MPI_`". The return code for successful completion is `MPI_SUCCESS`.

16

Communicator in MPI

- A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other.



17

Communicators

- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

18

Querying Information

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.
- The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

19

Our First MPI Program

```
/* hello_word */

#include <mpi.h>

main (int argc, char *argv[])
{
   int npes, my_rank;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &npes);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   printf("From process %d out of %d, Hello World!\n",
                                 my_rank, npes);
  MPI_Finalize();
}
```

hello_word.c

In S16-08 please check that you log in using **osprey** machine. Otherwise please do a remote access as follows:

ssh osprey1.cz3.nus.edu.sg

(You can use osprey1 to osprey24.)

20

Compilation (make sure that your current directory contains the MPI program or you have to do a change of directory (cd)):

mpicc hello_world.c

- an executable file 'a.out' is created

Execution (assume that the object code is in subdirectory mpi):

mpiexec -rsh -nopm -machinefile machine -n 5 $HOME/mpi/./a.out

```
osprey1.cz3.nus.edu.sg
osprey2.cz3.nus.edu.sg
osprey3.cz3.nus.edu.sg
osprey4.cz3.nus.edu.sg
osprey5.cz3.nus.edu.sg
osprey6.cz3.nus.edu.sg
osprey7.cz3.nus.edu.sg
osprey8.cz3.nus.edu.sg
osprey9.cz3.nus.edu.sg
osprey10.cz3.nus.edu.sg
osprey11.cz3.nus.edu.sg
osprey12.cz3.nus.edu.sg
osprey13.cz3.nus.edu.sg
osprey14.cz3.nus.edu.sg
osprey15.cz3.nus.edu.sg
osprey16.cz3.nus.edu.sg
osprey17.cz3.nus.edu.sg
osprey18.cz3.nus.edu.sg
osprey19.cz3.nus.edu.sg
osprey20.cz3.nus.edu.sg
osprey21.cz3.nus.edu.sg
osprey22.cz3.nus.edu.sg
osprey23.cz3.nus.edu.sg
osprey24.cz3.nus.edu.sg
```

'machine' is the file containing the name of machines to carry out the MPI jobs. This file has been sent to you in the combined zipped file.

'-n 5' set the number of processors to be 5. In this case the first 5 workstations are used.

'$HOME/mpi/./a.out' specifies the executable file with full pathname.

21

Output With Different Number of Processors Used

The screen output from all processors will be displayed on the screen of P0.

```
/* hello_word */

#include <mpi.h>

main (int argc, char *argv[])
{
  int npes, my_rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  printf("From process %d out of %d, Hello World!\n",
                                my_rank, npes);
 MPI_Finalize();
}
```

hello_word.c

Screen Output at P0

-n 1

```
From process 0 out of 1, Hello World!
```

-n 2

```
From process 0 out of 2, Hello World!
From process 1 out of 2, Hello World!
```

-n 5

```
From process 3 out of 5, Hello World!
From process 0 out of 5, Hello World!
From process 1 out of 5, Hello World!
From process 2 out of 5, Hello World!
From process 4 out of 5, Hello World!
```

22

MPI Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

23

Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively. These two are blocking functions.
- The calling sequences of these routines are as follows:

```
int MPI_Send(void *buf, int count, MPI_Datatype
      datatype, int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype
      datatype, int source, int tag,
      MPI_Comm comm, MPI_Status *status)
```

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons.
- The datatype `MPI_BYTE` corresponds to a byte (8 bits) and `MPI_PACKED` corresponds to a collection of data items that has been created by packing non-contiguous data.

24

Hello World Again by Send and Recv Functions

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

main(int argc, char **argv)
{

  char msg[20];
  int my_rank, tag =99;
  MPI_Status status;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  if (my_rank == 0){
    strcpy(msg, "Hello there");
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
    printf ("my_rank = %d: %s\n", my_rank, msg);
  }
  else if (my_rank == 1){
    MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("my_rank = %d: %s\n", my_rank, msg);
  }

  MPI_Finalize();
}
```

hello.c

25

Output With Different Number of Processors Used

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

main(int argc, char **argv)
{

  char msg[20];
  int my_rank, tag =99;
  MPI_Status status;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  if (my_rank == 0){
    strcpy(msg, "Hello there");
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
    printf("my_rank = %d: %s\n", my_rank, msg);
  }
  else if (my_rank == 1){
    MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("my_rank = %d: %s\n", my_rank, msg);
  }

  MPI_Finalize();
}
```

hello.c

Screen Output at P0

-n 2

my_rank = 1: Hello there
my_rank = 0: Hello there

-n 3

-n 4

-n 5

26

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

main(int argc, char **argv)
{

  char msg[20];
  int my_rank, tag =99;
  MPI_Status status;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  if (my_rank == 0){
    strcpy(msg, "Hello there");
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
    printf("my_rank = %d: %s\n", my_rank, msg);
  }
  else if (my_rank == 1){
    MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("my_rank = %d: %s\n", my_rank, msg);
  }

  MPI_Finalize();
}
```

hello.c

Screen Output at P0 (-n 1)

27

Sending and Receiving Messages

- MPI allows specification of wildcard arguments for both source and tag.
- If source is set to `MPI_ANY_SOURCE`, then any process of the communication domain can be the source of the message.
- If tag is set to `MPI_ANY_TAG`, then messages with any tag are accepted.
- On the receiver side, the message received must be of length equal to or less than the length field specified.

28

Sending and Receiving Messages

- On the receiving end, the status variable can be used to get information about the `MPI_Recv` operation.
- The corresponding data structure contains:

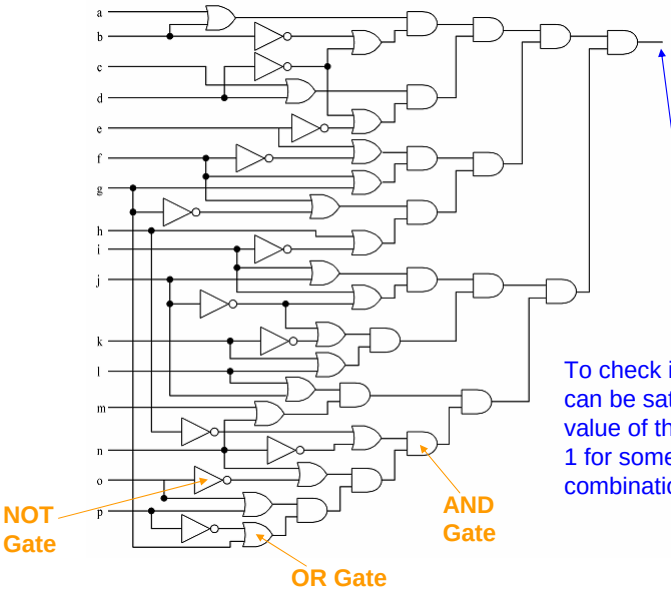
```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR; };
```

- The MPI_Get_count function returns the precise count of data items received.

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype
                  datatype, int *count)
```

29

Example: Circuit Satisfiability

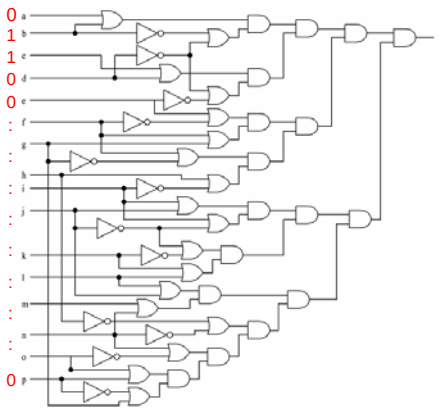


30

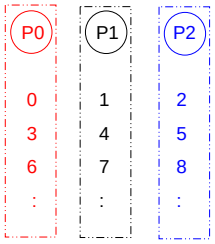
Solution Method

- Circuit satisfiability is a NP-complete problem.
- No known algorithms to solve in polynomial time.
- We seek all solutions.
- We find through exhaustive search.
- 16 inputs ⇒ 65,536 combinations to test. That's time consuming !!


```
/* sat1.c */

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

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

/* to be continued on next slide */
```

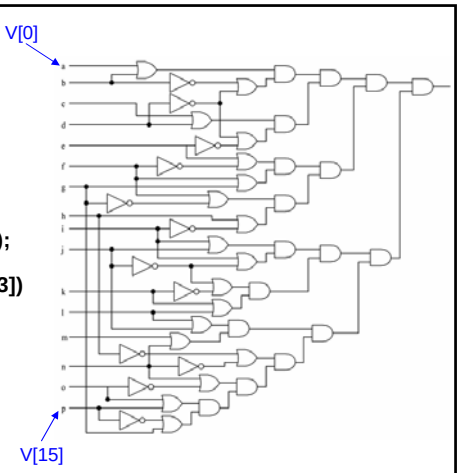


6



Test Cases in each Processors (Eg, *p*=3)

```
/* 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%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);
  }
}
```



33

sat1.c

```
MPI_Comm_rank (MPI_COMM_WORLD, &id);
        :
for (i = id; i < 65536; i += p)
    check_circuit (id, i);
```

Screen Output for sat1.c at P0

V[0] V[15]

-n 1

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

-n 2

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

-n 3

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

34

Modifications to find out the number of input combinations which satisfy the circuit

(sat2.c)

- These input combinations are scattered in various MPI processes. How to sum them together?
- Modify function **`check_circuit`**
  - Return 1 if circuit satisfiable with input combination
  - Return 0 otherwise
- Each process adds local count of satisfiable circuits it has found
- Perform reduction after **`for`** loop

35

New Declarations and Code

(sat2.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);  /* local sum*/
```

36

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 */
)
```

37

Predefined Reduction Operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

38

Our Call to MPI_Reduce( )

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

Only process-0 will get the result

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

39

```
/*
 *   Circuit Satisfiability, Version 2
 *
 *   This enhanced version of the program prints the total number of solutions.
 */

#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;
}

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

int 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%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);
      return 1;
   } else return 0;
}
```

Screen Output for sat2.c

-n 3

```
1) 1110111110011001
1) 1010111111011001
0) 0110111110011001
0) 1110111111011001
1) 0110111110111001
2) 1010111110011001
0) 1010111110111001
2) 0110111111011001
2) 1110111110111001
Process 1 is done
Process 2 is done
Process 0 is done
There are 9 different solutions
```

sat2.c

40

Benchmarking the Program

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

Function `MPI_Wtick` returns a double-precision floating-point number indicating the number of seconds between ticks of the clock used by function `MPI_Wtime`. For example, if the clock is incremented every microsecond, function `MPT_Wtick` should return the value 10^{-6}.

Function `MPI_Wtime` returns a double-precision floating-point number representing the number of seconds since some time in the past. The definition of "some time in the past " is guaranteed not to change during the life of a process. Hence the elapsed time of a block of code can be determined by calling `MPI_Wtime` before it and after it and computing the difference.

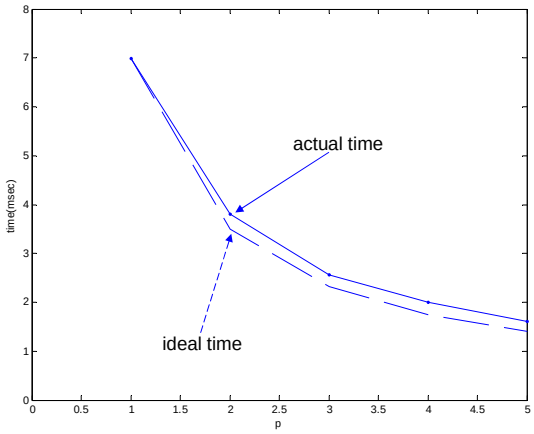
41

Benchmarking Code

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

42

```
/*
*   Circuit Satisfiability, Version 3
*
*   This version of the program prints the total number of  solutions
*   and the execution time.
*
*   Programmed by Michael J. Quinn
*
*   Last modification: 3 September 2002
*/

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

int main (int argc, char *argv[]) {
   int count;        /* Solutions found by this proc */
   double elapsed_time;  /* Time to find, count solutions */
   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);

   /* Start timer */
   MPI_Barrier (MPI_COMM_WORLD);
   elapsed_time = - MPI_Wtime();

   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);

   /* Stop timer */
   elapsed_time += MPI_Wtime();

   if (!id) {
      printf ("Execution time %8.6f\n", elapsed_time);
      fflush (stdout);
   }
   MPI_Finalize();
   if (id) printf ("There are %d different solutions\n",
      global_count);
   return 0;
}

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

int check_circuit (int id, int z) {
   int v[16];
        :
        :
   /* same */
}
```



Platform: CZ lab at S16-08
(osprey machines are used)

Avoiding Deadlocks

Consider:

```
int a[10], b[10], my_rank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (my_rank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

destination tag

source tag

Since MPI_Send is blocking, there will be a deadlock. **Why?**

Answer:

Avoiding Deadlocks

Consider the following piece of code, in which process i sends a message to process i + 1 (modulo the number of processes) and receives a message from process i - 1 (module the number of processes). **What is the effect of this code segment?**

```
int a[10], b[10], npes, my rank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &my rank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
       MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
       MPI_COMM_WORLD);
...
```

Process *i*

(*i* refers to my rank.)

Answers:

Avoiding Deadlocks

We can break the circular wait to avoid deadlocks as follows:

```
int a[10], b[10], npes, my rank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &my rank);
if (myrank%2 == 1) {  /* if I am of odd rank */
       MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
              MPI_COMM_WORLD);
       MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
              MPI_COMM_WORLD);
}
else {                /* if I am of even rank */
       MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
              MPI_COMM_WORLD);
       MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
              MPI_COMM_WORLD);
}
...
```

Process *i*

(*i* refers to my rank.)

Asymmetric Approach

Overlapping Communication with Computation

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations.

I stands for initiate

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

- These operations return before the operations have been completed. Function `MPI_Test` tests whether or not the non-blocking send or receive operation identified by its request has finished.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

- `MPI_Wait` waits for the operation to complete.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```


Example: Multiply Matrix (*n*x*n*) and Vector (*n*x1)

$$\begin{pmatrix}
28 & 79 & 20 & 52 & 87 & 17 & 48 & 26 & 86 & 98 & 46 & 74 & 28 & 18 & 67 & 55 \\
18 & 38 & 29 & 62 & 86 & 31 & 39 & 87 & 99 & 32 & 87 & 46 & 78 & 40 & 66 & 55 \\
66 & 49 & 14 & 96 & 62 & 2 & 100 & 86 & 13 & 5 & 44 & 45 & 99 & 78 & 4 & 36 \\
79 & 77 & 88 & 8 & 64 & 93 & 40 & 77 & 85 & 71 & 30 & 71 & 44 & 66 & 11 & 72 \\
77 & 64 & 6 & 63 & 46 & 83 & 9 & 35 & 37 & 16 & 3 & 48 & 63 & 93 & 73 & 57 \\
41 & 48 & 2 & 15 & 10 & 38 & 68 & 29 & 44 & 100 & 69 & 70 & 73 & 85 & 17 & 24 \\
96 & 9 & 45 & 84 & 22 & 64 & 43 & 96 & 20 & 64 & 94 & 83 & 15 & 39 & 15 & 99 \\
69 & 48 & 24 & 94 & 71 & 89 & 13 & 22 & 1 & 100 & 1 & 69 & 58 & 71 & 27 & 46 \\
18 & 95 & 59 & 73 & 90 & 98 & 99 & 13 & 48 & 48 & 31 & 42 & 55 & 2 & 82 & 1 \\
72 & 7 & 93 & 13 & 25 & 30 & 77 & 32 & 13 & 100 & 10 & 27 & 67 & 51 & 43 & 8 \\
3 & 1 & 83 & 26 & 48 & 14 & 41 & 45 & 77 & 58 & 1 & 34 & 53 & 48 & 98 & 86 \\
73 & 80 & 38 & 4 & 77 & 80 & 83 & 2 & 23 & 23 & 49 & 44 & 7 & 99 & 51 & 7 \\
3 & 44 & 69 & 72 & 16 & 18 & 82 & 18 & 92 & 59 & 78 & 25 & 41 & 85 & 48 & 75 \\
9 & 17 & 24 & 43 & 15 & 53 & 26 & 84 & 47 & 96 & 81 & 83 & 3 & 79 & 51 & 55 \\
54 & 85 & 76 & 27 & 69 & 46 & 36 & 100 & 38 & 84 & 1 & 21 & 61 & 3 & 75 & 12 \\
55 & 30 & 89 & 22 & 11 & 93 & 19 & 99 & 55 & 36 & 46 & 11 & 14 & 32 & 15 & 61
\end{pmatrix}
\times
\begin{pmatrix}
53 \\ 78 \\ 23 \\ 11 \\ 37 \\ 8 \\ 60 \\ 34 \\ 28 \\ 39 \\ 63 \\ 42 \\ 88 \\ 64 \\ 26 \\ 20
\end{pmatrix}
=
\begin{pmatrix}
34491 \\ 37668 \\ 39681 \\ 40278 \\ 32399 \\ 37073 \\ 33569 \\ 32013 \\ 35073 \\ 29983 \\ 26223 \\ 34458 \\ 34875 \\ 29618 \\ 33053 \\ 24395
\end{pmatrix}$$

No. of processes = (*p*-1) slaves + 1 master

So we allocate $\frac{n}{(p-1)}$ rows to each slave.

If *n* is not a multiple of (*p*-1), master process will multiply the remaining rows at the bottom with the vector.

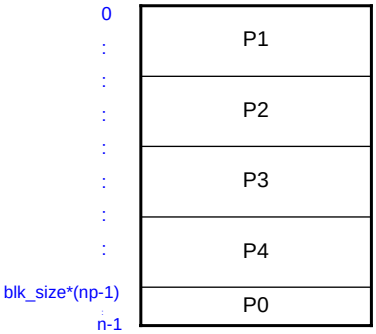
mpicc **–lm matrix_p.c**

```
#include <stdio.h>                    /*  matrix_p.c    */
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#define n 16
#define seed 5

void master (int np, int blk_size, int rem_size);
void slave (int np, int blk_size);

main(int argc, char **argv){
  int my_rank, np;
  int blk_size, rem_size;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &np);  /* np = (p-1) slaves + 1 master */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  blk_size = n/(np-1);                      // number of rows to be processed by each slave
  rem_size = n - blk_size*(np-1);    // rows left, processed by master itself
                                     // equal to 0 if n is a multiple of (np-1)
  if (my_rank == 0) master(np, blk_size, rem_size);
  else slave(np, blk_size);

  MPI_Finalize();
}                                  /* program continues on next slide */
```

49

```
void master (int np, int blk_size, int rem_size){
  int A[n][n], B[n], C[n];
  int i, j, sum;
  int proc;
  MPI_Status status;
  MPI_Request request;

  srand48(seed);            /* fix the seed of pseudo random number generator */

  for (i=0; i<n; i++){
     B[i] = 1+floor(drand48()*100);        /* drand48() gives 0 to 0.99999999... */
     for (j=0; j<n; j++)
        A[i][j] = 1+floor(drand48()*100);  /*  A[i][j] = 1, 2, 3, …, or, 99  */
  }

  for (proc=1; proc<np; proc++){
     MPI_Isend(A+blk_size*(proc-1), blk_size*n, MPI_INT, proc, 0, MPI_COMM_WORLD,
                                                                    &request);
     MPI_Send(B, n, MPI_INT, proc, 1, MPI_COMM_WORLD);
     MPI_Wait(&request, &status);          /* wait for Isend to complete */
  }

/*  Function master continues on the next slide */
```

50




```
  if (rem_size != 0){
    for (i=blk_size*(np-1); i<n; i++){
        sum = 0;
        for (j=0; j<n; j++)
            sum += A[i][j]*B[j];
        C[i] = sum;
    }
  }

  for (proc=1; proc<np ; proc++)
     MPI_Recv(C+blk_size*(proc-1), blk_size, MPI_INT, proc, 2,
    MPI_COMM_WORLD, &status);

  for (i=0; i<n; i++){
     for (j=0; j<n; j++)
       printf("%3d ", A[i][j]);
     printf("    %3d ", B[i]);
     printf("    %8d \n", C[i]);
  }
}
```

51

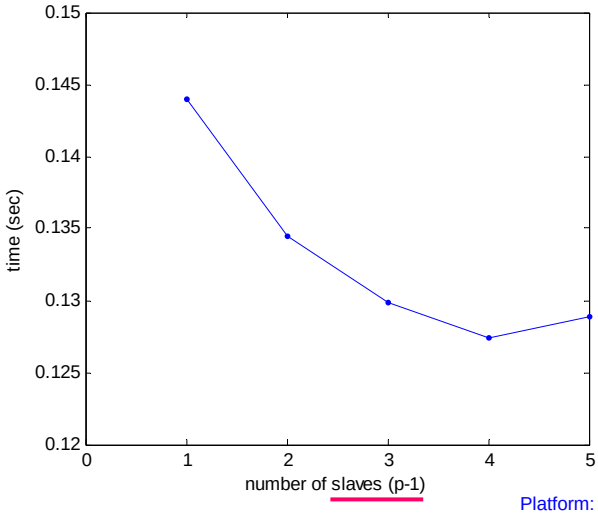
/* end of function master. Function slave continues on next slide */

```
void slave (int np, int blk_size){
  int B[n], C[n], *D;
  int i, j, sum;
  MPI_Status status1, status2;
  MPI_Request request;

  D=(int*)malloc(blk_size*n*sizeof(int));

  MPI_Irecv(D, blk_size*n, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
  MPI_Recv(B, n, MPI_INT, 0, 1, MPI_COMM_WORLD, &status1);
  MPI_Wait(&request, &status2);          /*  wait until Irecv is complete   */

  for (i=0; i<blk_size; i++){
     sum = 0;

     for (j=0; j<n; j++)
        sum += D[i*n+j]*B[j];

     C[i] = sum;
  }

  MPI_Send(C, blk_size, MPI_INT, 0, 2, MPI_COMM_WORLD);

}
```

52

/*end of program */

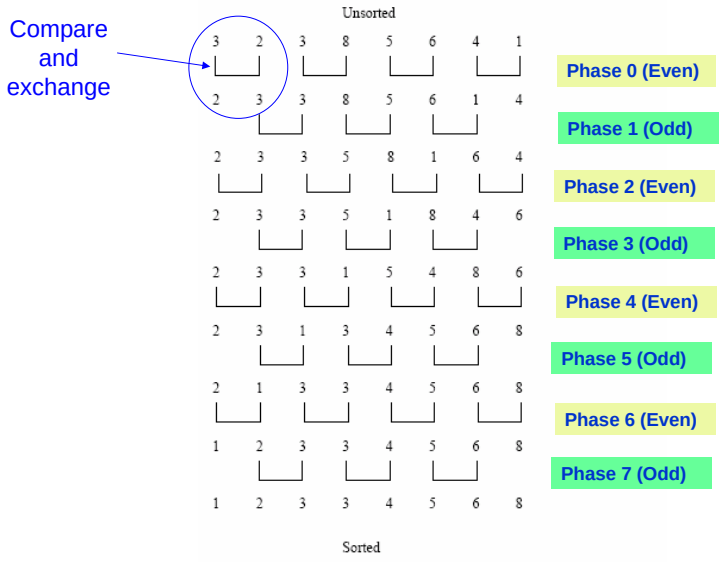
Performance of the MPI Matrix-Vector Multiplication Program (n=16)



Platform: CZ lab at S16-08

53

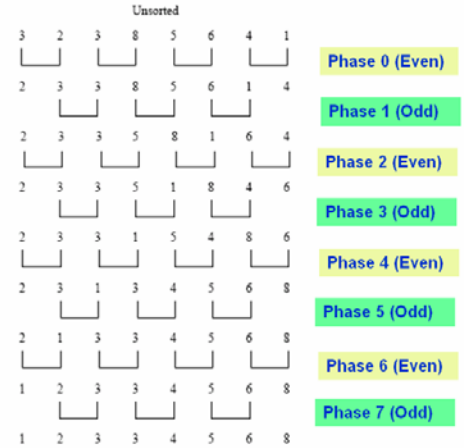
Example on Sorting: Even-Odd Transposition



Sorting $n = 8$ elements, using the even-odd transposition sort algorithm. During each phase, $n = 8$ elements are compared.

54

Even-Odd Transposition



- After n phases of odd-even exchanges, the sequence is sorted.
- Each phase of the algorithm (either odd or even) requires $\Theta(n)$ comparisons.
- Serial complexity is $\Theta(n^2)$.

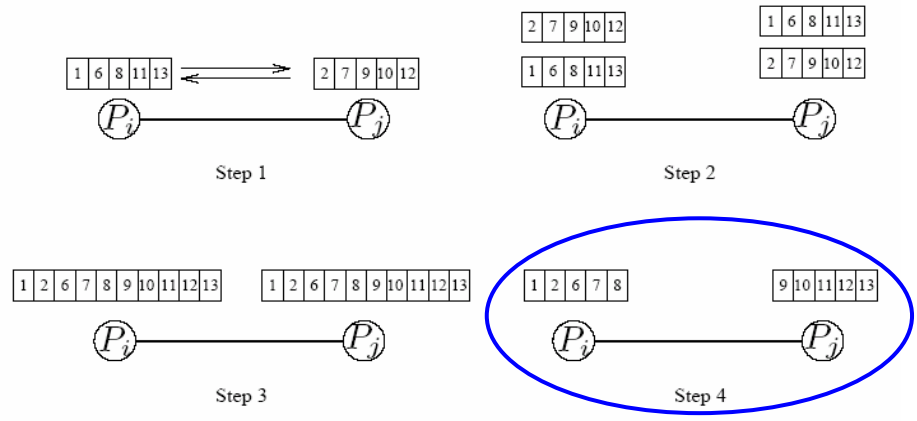
55

Parallel Even-Odd Transposition with n Processors each with 1 data

- Consider the one item per processor case. There are n iterations and in each iteration, each processor does one compare-exchange.
- The parallel run time of this formulation is $\Theta(n)$.
- This is cost optimal with respect to the base serial algorithm but not optimal when compared with quick sort.
- In practice we do not have n number of processors. In such cases $\frac{n}{p}$ data are allocated to a processor, and instead of doing the compare-and-exchange operation the processors perform compare-and-split operation.

56

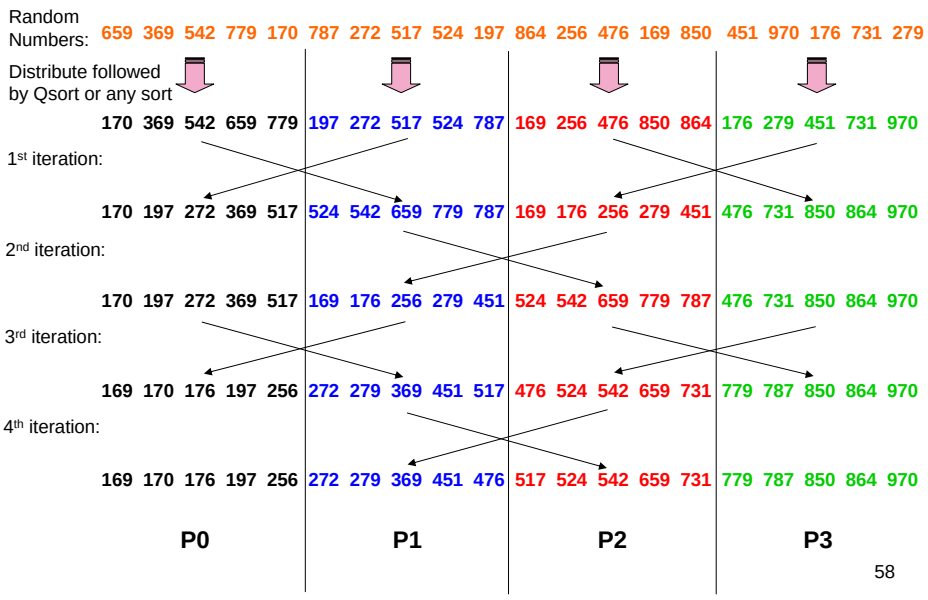
Compare and Split Operation



In compare and split operation, each process sends its block of size *n/p* to the other process (steps 1 and 2). Subsequently, each process merges the received block with its own block and retains only the appropriate half of the merged block (steps 3 and 4). In this example, process P_i retains the smaller elements and process P_i retains the larger elements.

57

Parallel Even-Odd Sort



58

QSORT(3) Linux Programmer's Manual QSORT(3)

NAME
qsort - sorts an array

SYNOPSIS
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
int(*compar)(const void *, const void *));

DESCRIPTION
The qsort() function sorts an array with nmemb elements of size size. The base argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by compar, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

RETURN VALUE
The qsort() function returns no value.

CONFORMING TO
SVID 3, POSIX, 4.3BSD, ISO 9899

59

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <assert.h>

static int cmpstringp(const void *p1, const void *p2)
{
   /* The actual arguments to this function are "pointers to
      pointers to char", but strcmp() arguments are "pointers
      to char", hence the following cast plus dereference */

   return strcmp(* (char * const *) p1, * (char * const *) p2);
}

int  main(int argc, char *argv[])
{
   int j;

   assert(argc > 1);

   qsort(&argv[1], argc - 1, sizeof(char *), cmpstringp);

   for (j = 1; j < argc; j++)
       puts(argv[j]);
   exit(EXIT_SUCCESS);
}
```

Built-in Quick Sort Example

60

Sending and Receiving Messages Simultaneously to facilitate Compare-and-Split Operation

To exchange (send and receive) messages, MPI provides the following function:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
   MPI_Datatype senddatatype, int dest, int
   sendtag, void *recvbuf, int recvcount,
   MPI_Datatype recvdatatype, int source, int recvtag,
   MPI_Comm comm, MPI_Status *status)
```

The arguments include arguments to the send and receive functions. If we wish to use the same buffer for both send and receive, we can use:

```
int MPI_Sendrecv_replace(void *buf, int count,
   MPI_Datatype datatype, int dest, int sendtag,
   int source, int recvtag, MPI_Comm comm,
   MPI_Status *status)
```

61

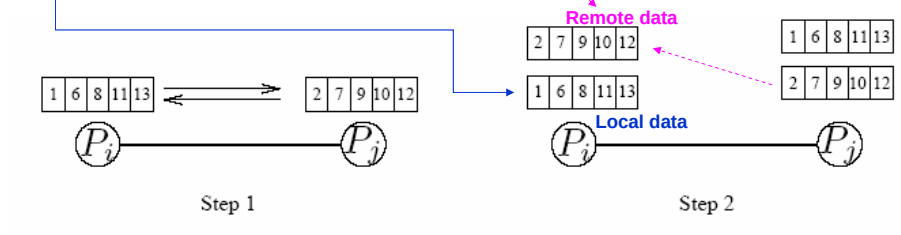
Parallel Even-Odd Sort

even_odd.c

```
#include <stdlib.h>
#include <mpi.h> /* Include MPI's header file */

#define seed 5

int IncOrder(const void *e1, const void *e2);

main(int argc, char *argv[])
{
  int n;          /* The total number of elements to be sorted */
  int npes;       /* The total number of processes */
  int myrank;     /* The rank of the calling process */
  int nlocal;     /* The local number of elements, and the array that stores them */
  int *elmnts;    /* The array that stores the local elements */
  int *relmnts;   /* The array that stores the received elements */
  int oddrank;    /* The rank of the process during odd-phase communication */
  int evenrank;   /* The rank of the process during even-phase communication */
  int *wspace;    /* Working space during the compare-split operation */
  int i, j;
  MPI_Status status;
  int sum, reduce_sum;

  /* Initialize MPI and get system information */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

62

```
n = atoi(argv[1]);
nlocal = n/npes; /* Compute the number of elements to be stored locally. */

/* Allocate memory for the various arrays */
elmnts  = (int *)malloc(nlocal*sizeof(int));
relmnts = (int *)malloc(nlocal*sizeof(int));
wspace  = (int *)malloc(nlocal*sizeof(int));
```

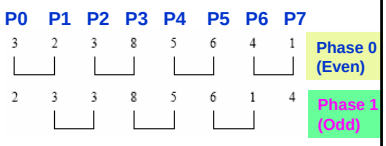


63

```
sum = 0;

if (myrank == 0){
  /* Fill-in the elmnts array with random elements */
  srand48(seed);

  for (j=1; j<npes; j++){   /* send out random number to (p-1) processes */
    for (i=0; i<nlocal; i++)
      elmnts[i] = floor(drand48()*999);
          MPI_Send(elmnts, nlocal, MPI_INT, j, 0, MPI_COMM_WORLD);
  }

  for (i=0; i<nlocal; i++){      /* assign random number to local process */
    elmnts[i] = floor(drand48()*999);
    sum += elmnts[i];
  }
}
else{      /* receive the random numbers */
          MPI_Recv(elmnts, nlocal, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

          for(i=0; i<nlocal; i++)
            sum += elmnts[i];
}
```

64

```
MPI_Reduce(&sum, &reduce_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
 if(myrank == 0)
          printf("\nsum before sorting = %d \n\n", reduce_sum);

 /* Sort the local elements using the built-in quicksort routine */
 qsort(elmnts, nlocal, sizeof(int), IncOrder);  /* each process sorts the local list */

 printf("\nIteration 0, id = %d: ", myrank);
 for( i=0; i<nlocal; i++ )
          printf("%4d ", elmnts[i]);

 /* Determine the rank of the processors that myrank needs to communicate during the */
 /* odd and even phases of the algorithm */
 if (myrank%2 == 0) {
  oddrank  = myrank-1;       /* odd phase partner for even ID*/
  evenrank = myrank+1; /* even phase partner for even ID*/
 }
 else {
  oddrank  = myrank+1;       /* odd phase partner for odd ID*/
  evenrank = myrank-1; /* even phase partner  for odd ID*/

 }

 /* Set the ranks of the processors at the end of the linear */
 if (oddrank == -1 || oddrank == npes)
   oddrank = MPI_PROC_NULL;
 if (evenrank == -1 || evenrank == npes)
   evenrank = MPI_PROC_NULL;
```

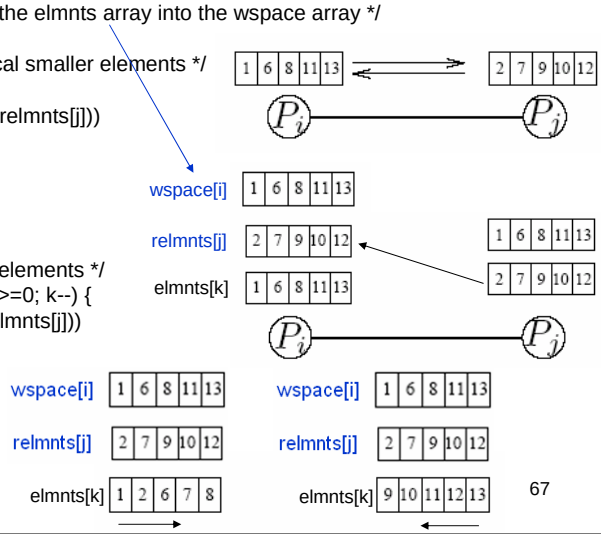


65

```
 /* Get into the main loop of the odd-even sorting algorithm */
    for (i=0; i<npes; i++) {
      if (i%2 == 1) /* Odd phase , send/receive from odd partner */
        MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
           nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
      else /* Even phase, send/receive from even partner */
        MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
           nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);

      if (status.MPI_SOURCE >= 0)
        CompareSplit(nlocal, elmnts, relmnts, wspace, myrank < status.MPI_SOURCE);

      printf("\nIteration %d, id = %d: ", i+1, myrank);
      for( j=0; j<nlocal; j++ ) printf("%4d ", elmnts[j]);
    }

    printf("\n");

    sum = 0;

    for (i=0; i<nlocal; i++) sum += elmnts[i];

    MPI_Reduce(&sum, &reduce_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myrank == 0)
      printf("\nsum after sorting = %d \n\n", reduce_sum);

    free(elmnts); free(relmnts); free(wspace);
    MPI_Finalize();
  }
```

66

```
/* This is the CompareSplit function */
  CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace, int keepsmall)
  {
    int i, j, k;

    for (i=0; i<nlocal; i++)
      wspace[i] = elmnts[i]; /* Copy the elmnts array into the wspace array */

    if (keepsmall) { /* Keep the nlocal smaller elements */
      for (i=j=k=0; k<nlocal; k++) {
        if ((i < nlocal && wspace[i] < relmnts[j]))
          elmnts[k] = wspace[i++];
        else
          elmnts[k] = relmnts[j++];
      }
    }
    else { /* Keep the nlocal larger elements */
      for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
        if ((i >= 0 && wspace[i] >= relmnts[j]))
          elmnts[k] = wspace[i--];
        else
          elmnts[k] = relmnts[j--];
      }
    }
  }
```




```
/* The IncOrder function that is called by qsort is defined as follows */

int IncOrder(const void *e1, const void *e2)
{
  return (*((int *)e1) - *((int *)e2));
}
```

even_odd.c

Other MPI Instructions

Please refer to the handouts.