Solutions For Selected Exercises In: Parallel Programming with MPI by Peter S. Pacheco

John L. Weatherwax[*]

November 4, 2006

Additional Notes and Derivations

Physical Constraints on Serial Computers (Page 4)

The speed of light is $c = 3\,10^8$ m/s and the code given must execute 3 (one for each component of `x`, `y`, and `z`) trillion memory transfers each second. Thus the transfer flux out of memory is $n = 3\,10^{12}$transfers/s. From elementary physics

$$\text{distance} = \text{rate} \times \text{time} \tag{1}$$

Thus if r is the average distance from a single memory location to the CPU in one second on a serial machine we travel a *total* distance of $r \times n$. Then Eq. 1 gives the following expression for r

$$r = \frac{tc}{n} = \frac{1 \times 3\,10^8}{3\,10^{12}} = 10^{-4}\text{m} \tag{2}$$

As suggested in the book, placing our CPU in the center of a square grid with side length s, the average distance to each memory location is $r = s/2$. From the above this gives a linear dimension of $s = 2\,10^{-4}$m. Since we are assuming that all three trillion memory modules are inside this square, the number of memory modules along any given linear dimension would be

$$n_m = \sqrt{3\,10^{12}} = \sqrt{3}\,10^6 \tag{3}$$

Based on the previous length estimate of a side of $s = 2\,10^{-4}$m we see that the physical length of each memory module m_l must satisfy

$$\begin{aligned} n_m m_l &= s \qquad \text{or} & (4)\\ \sqrt{3}\,10^6 m_l &= 2\,10^{-4} & (5) \end{aligned}$$

Solving for m_l we obtain $m_l = \frac{2}{\sqrt{3}}10^{-10}\text{m} \approx 1\,\text{angstrom}$. Clearly an impossible situation.

[*]wax@alum.mit.edu

Problem Solutions

Chapter 1 (Introduction)

Chapter 1 had no problems.

Chapter 2 (An Overview of Parallel Computing)

Exercise 1

Part (a) In store and forward routing each node must store the entire message before it gets passed on to the next node in the transmission. Thus assuming that one packet can be transmitted per timestep it will require $O(n)$ timesteps to transmit this message to each node. For k intermediate nodes we have $k+1$ "edges" in our connection graph giving a total transmission time of $O((k+1)n) = O(nk)$

Part (b) Using cut-through routing each intermediate node can send any packet of a message to the next host as it is received. Thus assuming host A is sending to host B, the first timestep will have one packet from A to the first intermediate node between A and B. In two timesteps packets will have propagated two the first two intermediate nodes. Thus in $k+1$ timesteps we will have packets arriving at B. If $k < n$ we require $O(n)$ timesteps to transmit our message. If $k > n$ we require $O(k)$ timesteps to transmit our message.

Exercise 2

Shared memory programming has three basic primatives:

- Variables can be accessed by all processors
- There exists a means to prevent improper access of shared resources (via binary semaphores or some other means)
- There exists a means for synchronizing the processes (via barriers).

To solve this problem consider the odd-even sorting algorithm which can sort in $O(n)$ steps, see [1] for more details. In this sorting algorithm, during odd-numbered steps the odd-numbered processors compare their number with that of their next higher numbered even processor and exchange if the two numbers are out of sequence. During even numbered steps the even-numbered processor compare their number with that of their next higher odd-numbered processor. A pseudo-code implementation (with care to avoid deadlock in critical regions) is given by the following

```
ODD-EVEN SORT($a, n$)
for $i \leftarrow 1$ to n
      do
         if $i$ mod $2 = 1$
            then
                  ▷ $i$ is an odd-timestep
                  if $p$ mod $2 = 1$
                     then
                           ▷ processor $p$ is an odd-processor
                           Lock array elements $a[p]$ and $a[p+1]$
                           Sort elements $a[p]$ and $a[p+1]$
                           Insert back into global array $a$ in sorted order
                           Unlock array elements $a[p]$ and $a[p+1]$
                     else
                           ▷ Do nothing

            else
                  ▷ $i$ is an even-timestep
                  if $p$ mod $2 = 0$
                     then
                           ▷ processor $p$ is an even-processor
                           Lock array elements $a[p]$ and $a[p+1]$
                           Sort elements $a[p]$ and $a[p+1]$
                           Insert back into global array $a$ in sorted order
                           Unlock array elements $a[p]$ and $a[p+1]$
                     else
                           ▷ Do nothing
```

Exercise 3

If we ran the given pseudocode on a large number of processors, depending on the scheduling of the processes some process may never execute the critical section/region. In other words some subset of the processors may lock the binary semaphore `s` for all time not allowing access to the complementary subset of processors. The effect is to exclude some of the processors from the critical section.

Exercise 4

Part (a): WWX: Finish me!!!

Chapter 3 (Greetings!)

Exercise 1

See the code `prob_3.6.1.c`. When this code is run on only one processor no output is produced.

Exercise 2

See the code `prob_3.6.2.c`. When using wildcards for in the receives I didn't get any noticeable difference in output, which is expected since the code is issuing it `MPI_Recv` calls in a particular order and thus blocks until it receives each message before printing.

Exercise 3

Please see the code `prob_3.6.3.c`. I experimented with the following modifications to the calls to `MPI_Send` and `MPI_Recv`.

- Covert the destination to 1 in all sending process in order to test incorrectly matched `MPI_Send` and `MPI_Recv` calls. This results in the program hanging forever since `MPI_Recv` blocks and is never able to complete.
- Execute `MPI_Send` with an incorrect string length by removing the required +1 from the `MPI_Send` call. The results of this modification were that the program still worked but the executed `printf` call will print characters until it encounters the first terminating null located randomly in memory.
- Specifying an incorrect MPI data type in the `MPI_Send` call only. For instance specifying `INT` rather than `CHAR` causes the code to crash.
- Specifying an incorrect receive size of 10 rather than the correct value of 100 resulted in the code crashing.
- Specifying an incorrect MPI data type in the `MPI_Recv` call. For instance specifying `INT` rather than `CHAR` resulted in a program that seemed to execute correctly.
- Specifying an incorrect tag field in the `MPI_Recv` call results in the programming hanging since it waits forever for messaging passing to complete.

Exercise 4

See the code `prob_3.6.4.c`. On my system the process $p-1$ could print to the screen. Printing on any processor other than 0 is not required by an MPI implementation however.

Programming Assignment 1

See the code `prob_3.7.1.c`. Calculating who to send a message to is simple and is given by

```
  dest = (my_rank+1) % p;
```

as suggested in the text. Calculating who to receive a message from is done with code like the following

```
  recv = ( (my_rank==0) ? (p-1) : (my_rank-1) );
```

where we have been careful to correctly specify that the *first* process sends to the *last* process. Each process must send its message first and then receive. In the other order each process hangs waiting for messages that never arrive. In the source code coming from this problem we see coded another message sending strategy where the even processors send first and then receive while the odd processor receive first and then send. This message scheduling works as well.

When run on one processor, processor 0 sends and receives a message from itself.

Chapter 4 (An Application: Numerical Integration)

Exercise 1

See the code `prob_4.6.1.c`. When run on one processor the code gives the correct result of 1/3, since in that case the local integration is equivalent to the global integration.

Exercise 2

See the code `prob_4.6.2.c`. The routine `Get_data` should be called before the individual processors integration domain. No modification besides including `Get_data` are required to implement this program.

Programming Assignment 1

See the code `prob_4.7.1.c`. The most complicated part of this problem is the specification of the choice of functions in which a user could choose to integrate. This was done by specifing an array of function pointers (a `fn_array`) with the command

```
  float (*fn_array[])(float) = {f1,f2,f3};
```

the user only then has to input an integer specifying the function to be integrated. Much more complicated menuing systems could be considered.

Programming Assignment 2

Part (a): See the code `prob_4.7.2.a.c`, where a serial version of Simpson's rule is implemented.

Part (b): See the code `prob_4.7.2.b.c`, where a parallel version of Simpson's rule is implemented.

Chapter 5 (Collective Communication)

Exercise 1

WWX: Finish!!!

Exercise 2

When this section of code is executed each processor executes its corresponding block of commands. As such, each processor begins by executing a `MPI_Bcast` statement. Since the *root* argument for all of these `MPI_Bcast` calls is 0 all processors update their *variable* argument based on that which is sent from processor 0. As coded, processor 0 is "sending" the variable `x`, processor 1 is "receiving" the variable `x`, and processor 2 is "receiving" the variable `z`. After the first `MPI_Bcast` call we have

- Process 0 with no change to the variable `x` giving $x = 0$
- Process 1 with an updated variable `x` giving $x = 0$
- Process 2 with an updated variable `z` giving $z = 0$

After each process has finished its calls to `MPI_Bcast` process 0 and 2 must execute an `MPI_Send` and an `MPI_Recv` respectively, while processor 1 must execute a collective communication `MPI_Bcast`. Since the `MPI_Bcast` acts as a synchronization point in the subsequent processing process 1 must wait until the other processes call `MPI_Bcast` themselves. Thus the `MPI_Send` and `MPI_Recv` on process 0 and 2 causes the variable x on process 2 to become the value of the variable y on process 0, or the numerical value of 1.

After this exchange, each processor calls (or has called in the case of processor 1) a `MPI_Bcast` routine. We can analyze this exchange in the same way that as for the first example of the global communication primitive `MPI_Bcast`. Since the *root* argument for all of these `MPI_Bcast` calls is 1 all processors update their *variable* argument based on that which is sent from processor 1. As coded, processor 1 is "sending" the variable `y`, processor 0 is "receiving" the variable `z`, and processor 2 is "receiving" the variable `y`. Thus after the completion of this `MPI_Bcast` call we have caused the following updates

- Process 0 has `z` updated with the value of `y` in process 1 giving $z = 4$
- Process 1 has the variable `y` unchanged giving $y = 4$
- Process 2 has `y` updated with the value of `y` in process 1 giving $y = 4$

Keeping track of all the variable values after all communication calls we have the state of the system of

Process 0	Process 1	Process 2
$x = 0$	$x = 0$	$x = 1$
$y = 1$	$y = 4$	$y = 4$
$z = 4$	$z = 5$	$z = 0$

Exercise 4

On process 0, the sequential `MPI_Send` calls access the following data structures/elements in this order:

x, second row of matrix B, x, fourth column of matrix B, first column of matrix B

Similarly on process 1, the sequential `MPI_Recv` calls access the following data structures/elements on process 1 in this order:

x, second row of matrix B, x, second column of matrix B, first column of matrix B.

Chapter 7 (Communicators and Topologies)

Exercise 1

Part (a): See the code `prob_7.11.1.a.c`. Rather than create a communicator associated with the processors in the first *column* of a virtual grid of processors the program `prob_7.11.1.a.c` creates a communicator associated with an input *row* index (zero based) of the virtual grid of processors. The modification to perform the requested column based communicator (using `MPI_Comm_group`, `MPI_Group_incl`, etc.) is straightforward.

Part (b): See the code `prob_7.11.1.b.c`. There we use `MPI_Comm_split` to create n communicators, broadcast a value of 1 along each column, and then use `MPI_Reduce` to compute the global sum.

Part (c): I would think that the processors *would* be identical since in both MPI calls our implicit assumption is that the global processors $0, n, 2n, 3n, \ldots$ would be associated with the first column.

Exercise 2

Part (a): In a call to `MPI_Comm_create` we would have to first construct a unique integer representing the new communicators context. This would entail looking at each process in the group to be created and determing an integer that is unique among all of the existing contexts already held by the processors included in this new communicator. This would entail *global* communication among processors.

Part (b): In a call to `MPI_Comm_split` we would use the input argument `split_key` to construct the associated communicator array. In addition, to the implementation of `MPI_Comm_create` our implementation of `MPI_Comm_split` would then have to find a unique integer to represent the communicator's context. This could be performed as above.

Exercise 3

In the modified basic algorithm given in the book we distributed our matrices in a block checkerboard fashion along the processors. In this problem we are to modify this basic algorithm (where each processor stores only a single element from each matrix) to the situation where each processor will store a block of rows from each matrix, specifically if we assume that n (the size of our square matrix) is divisible by p (the number of processors) then each processor will store n/p rows. A version of Fox's algorithm with this data distribution might be given by

```
BlockRowFox($n, p$)
1  for $q \leftarrow 0$ to $n/p - 1$
2      do
3          ▷ Gather columns $q\frac{n}{p}$ to $(q+1)\frac{n}{p} - 1$ onto each processor
4          ▷ Locally multiply all of my rows by the newly obtained columns
```

This modified version Fox's algorithm requires storage $O(2\frac{n}{p}n)$ for each processors share of the global matrices rows. The 2 is for storage for both matrix A and B. In addition, after a gather statement each processor will require an addition amount of storage given by $O(\frac{n}{p}n)$ to store the newly obtained columns. Finally, after multiplication each processor will have to store the C matrix requiring an additional $O(\frac{n}{p}n)$ storage. In total this modified version of Fox's algorithm requires $O(4\frac{n^2}{p})$ storage.

In a similar, way the block checkerboard basic algorithm requires $O(4\frac{n^2}{p^2})$ storage.

From these two results we see a trade off between memory usage and required message passing. The block checkerboard algorithm requires less memory but at the cost of more message passing (the broadcast of a specific matrix at each timestep), while the modified Fox's algorithm requires more storage but fewer actual sent messages (since messages are only needed in performing the column gather).

Exercise 4

The program discussed in this exercise would use a call like the following to construct the original Cartesian coordinate grid

```
dim_sizes[0] = l; dim_sizes[1] = m; dim_sizes[2] = n;
wrap_around[0] = 0; wrap_around[1] = 0; wrap_around[2] = 0;
MPI_Cart_create(MPI_COMM_WORLD,3,dim_sizes,wrap_around,0,grid_comm);
```

In the above we have not considered a periodic grid, and we have not allowed the underlying MPI implementation to reorder the global processors when creating this communicator.

Part (a): To create the desired communicator one would execute something like

```
free_coords[0] = 1;
free_coords[1] = 0;
free_coords[2] = 1;
MPI_Cart_sub(grid_comm,free_coords,&part_a_comm);
```

Part (b): To create the desired communicator one would execute something like

```
free_coords[0] = 0;
free_coords[1] = 0;
free_coords[2] = 1;
MPI_Cart_sub(grid_comm,free_coords,&part_b_comm);
```

Part (c): To create the desired communicator one would execute something like

```
free_coords[0] = 0;
free_coords[1] = 0;
free_coords[2] = 0;
MPI_Cart_sub(grid_comm,free_coords,&part_c_comm);
```

I would *not* assume that the communicator defined on process 0 is the same as the communicator defined in part c (above).

Exercise 5

As suggested in the text we can implement a safe circular shift of data using only `MPI_Send` and `MPI_Recv` (that will work if there is no buffering) if we take care to issue our send and recives in a certain manner. In case one is working on a system which does not provide buffering we can have the even processors issue `MPI_Send` and the odd processors issue the `MPI_Recv` second. A piece of code that demonstrates this is given below (where we are sending a string message from processor to processor)

```
  if( my_rank % 2 == 0 ){
  /* Use strlen+1 so that '\0' gets transmitted */
  printf("Process %d sending: %s\n", my_rank, messageS);
  MPI_Send(messageS, strlen(messageS)+1, MPI_CHAR,
   dest, tag, MPI_COMM_WORLD);

  MPI_Recv(messageR, 100, MPI_CHAR, recv, tag,
   MPI_COMM_WORLD, &status);
  printf("Process %d recieved: %s\n", my_rank, messageR);
}else{
  MPI_Recv(messageR, 100, MPI_CHAR, recv, tag,
   MPI_COMM_WORLD, &status);
  printf("Process %d recieved: %s\n", my_rank, messageR);

  /* Use strlen+1 so that '\0' gets transmitted */
  printf("Process %d sending: %s\n", my_rank, messageS);
  MPI_Send(messageS, strlen(messageS)+1, MPI_CHAR,
   dest, tag, MPI_COMM_WORLD);
}
```

In general, I believe that many MPI systems do provide some amount of buffering so that the circular shift discussed here can be coded and will work if the `MPI_Send`'s are issued *first*. In fact `prob_3.7.1.c` was first implemented in that manner and later coded to be made safe.

Programming Assignment 1

Programming Assignment 2

Chapter 10 (Design and Coding of Parallel Programs)

Exercise 2

I would have each processor seed the random number generator with its process rank. This way the random numbers would be guaranteed to at least be different in each processor. Code like the following should work

```
int p;
MPI_Comm_rank(MPI_COMM_WORLD, &p);
srand48( (long int) p);
```

Where the cast is required by the input arguments of `srand48`.

Programming Assignment 3

Because the MPI standard prohibits argument aliasing, if a program calls `MPI_Alltoall` correctly it will need to have two matrix variables declared. One to represent the original matrix and the other representing its transpose. In the implementation of this problem provided here we loop over the rows of `A` calling `MPI_Alltoall` and receiving its results in a temporary variable. We then copy the data into the additional storage representing the transposed matrix. We could not copy this transpose data back into the original matrix `A` or we would overwrite needed elements on the rows yet to be seen.

In many matrix algorithms (with block distribution of the rows among the processors) when `p` does not evenly divide `n` the following simple modification permit the use of non evenly divisible matrices by placing all "spill over" elements in the processor "p-1" (the last processor). This can be accomplished with the following code snippet

```
int n_bar, n_rem, n_local;
n_bar = n / p; /* using C's integer (truncated division) */
```

```
  n_rem = n % p; /* computes the remainder */
  n_local = ( my_rank != p-1 ? n_bar : n_bar+n_rem );
```

Then the computation in each processor proceeds by looping over the `n_local` rows as normal.

Chapter 11 (Performance)

Exercise 1

Some searching algorithms can have superlinear speedup. This is because in general the more processors one has the more of a give data structure that can be searched. This in tern can produce very quick query times which can (in some cases) result in superlinear speedup.

References

[1] J. R. Smith. *The design and analysis of parallel algorithms.* Oxford University Press, Inc., New York, NY, USA, 1993.