HPSC assignment compilation

Soumick Pyne 18AE10027

---

1. **Jacobi.cpp**

```
#include<iostream>
#include<stdio.h>
#include<conio.h>
#include<math.h>

using namespace std;

int main()
{
	int ans = 1;
	while (ans != 0)
	{
		int i, j;
		double l = 0;
		const int n = 3;
		double a[n][n], b[n][1], x[n][1], T[n][1], e, k;
		cout << "[a].[x]=[b]" << endl;
		cout << "Enter Matrix a:" << endl;
		for (i = 0; i < n; i++)
			for (j = 0; j < n; j++)
			{
				cout << "a[" << i << "," << j << "] = ";
				cin >> a[i][j];
			}
		cout << "Enter Matrix b:" << endl;
		for (i = 0; i < n; i++)
		{
			cout << "b[" << i << "][0] = ";
			cin >> b[i][0];
		}
		cout << "Enter the Accuracy = ";
		cin >> e;
		for (i = 0; i < n; i++)
			T[i][0] = 0;
		l = 0;
		while (1)
		{
			cout << "\nIteration no: " << l << "\n";
```

```
		for (i = 0; i < n; i++)
		{
			x[i][0] = (1 / a[i][i])*(b[i][0]);
			//cout << x[i][0] << " ";
			for (j = 0; j < n; j++)
			{
				if (j != i)
					x[i][0] = x[i][0] - (1 / a[i][i])*(a[i][j] * T[j][0]);
				//cout << x[i][0] << " ";
			}
			//cout << "\n";
		}
		k = 0;
		for (i = 0; i < n; i++)
		{
			k += pow(fabs(x[i][0] - T[i][0]), 2);
		}
		if (k <= e)
		{
			break;
		}
		l++;
		for (i = 0; i < n; i++)
			T[i][0] = x[i][0];

		/*for (i = 0; i < n; i++)
			cout << "x" << i + 1 << "=" << x[i][0] << endl;*/
	};
	for (i = 0; i < n; i++)
		cout << "x" << i + 1 << "=" << x[i][0] << endl;

	//trying spectral radius calculation
	double p_G = 0, sum1 = 0, sum2 = 0, base = 0, expo = 0;
	for (i = 0; i < n; i++)
	{
		sum1 += (x[i][0] - T[i][0])*(x[i][0] - T[i][0]);
	}
	cout << sum1 << "\n";
	for (i = 0; i < n; i++)
	{
		sum2 += x[i][0] * x[i][0];
	}
	cout << sum2 << "\n";
	base = sum1 / sum2;
	cout << " base = " << base;
	expo = 1 / l;
```

```
                cout << "expo = " << expo;
                p_G = pow(base, expo);
                cout << "\nspectral radius of G matrix = " << p_G;
                cout << "\ncontinue (1 or 0)?";
                cin >> ans;
        };
        _getch();
        return 0;
}
```

Result

Please ignore the spectral radius calculation.

```
soumick@soumick-HP-Pavilion-Laptop-15-cc1xx:~/hpsc_old_codes$ g++ jacobi.cpp
soumick@soumick-HP-Pavilion-Laptop-15-cc1xx:~/hpsc_old_codes$ ./a.out
[a].[x]=[b]
Enter Matrix a:
a[0,0] = 6
a[0,1] = 2
a[0,2] = 0
a[1,0] = 0
a[1,1] = 12
a[1,2] = 2
a[2,0] = 0
a[2,1] = 0
a[2,2] = 4
Enter Matrix b:
b[0][0] = 4
b[1][0] = 7
b[2][0] = 2
Enter the Accuracy = 0.00001

Iteration no: 0

Iteration no: 1

Iteration no: 2

Iteration no: 3
x1=0.5
x2=0.5
x3=0.5
```

The code with IOFF provision was used in midsem but I didn't upload the code to my drive after the exam.

2. Seidel.cpp

```
#include<iostream>
#include<stdio.h>
#include<conio.h>
#include<math.h>

using namespace std;

int main()
{
	int ans = 1;
	while (ans != 0)
	{
		int i, j;
		double l = 0, k = 0;
		//const int n = 200;
		const int n = 3;
		double a[n][n], b[n][1], x[n][1], T[n][1], T_dash[n][1], e;
		cout << "[a].[x]=[b]" << endl;
		cout << "Enter Matrix a:" << endl;
		for (i = 0; i < n; i++)
			for (j = 0; j < n; j++)
			{
				cout << "a[" << i << "," << j << "] = ";
				cin >> a[i][j];
			}
		cout << "Enter Matrix b:" << endl;
		for (i = 0; i < n; i++)
		{
			cout << "b[" << i << "][0] = ";
			cin >> b[i][0];
		}
		/*for (i = 0; i < n; i++)
			for (j = 0; j < n; j++)
			{
				if (i == j)
					a[i][j] = 200;
				else
					a[i][j] = 1;
			}
		for (i = 0; i < n; i++)
		{
			b[i][0] = 299;
		}*/
		e = 0.000001;
```

```
for (i = 0; i < n; i++)
{
	T[i][0] = 0;
	T_dash[i][0] = 0;
}
l = 0;
while (1)
{
	cout << "\nIteration no: " << l << "\n";
	for (i = 0; i < n; i++)
	{
		x[i][0] = (1 / a[i][i])*(b[i][0]);
		//cout << x[i][0] << " ";
		for (j = 0; j < n; j++)
		{
			if (j != i)
				x[i][0] = x[i][0] - (1 / a[i][i])*(a[i][j] * T[j][0]);
			//cout << x[i][0] << " ";
		}
		//cout << "\n";
		T_dash[i][0] = T[i][0];
		T[i][0] = x[i][0];
	}
	k = 0;
	for (i = 0; i < n; i++)
	{
		k += pow(fabs(x[i][0] - T_dash[i][0]), 2);
	}
	if (k <= e)
	{
		break;
	}
	l++;

};
for (i = 0; i < n; i++)
	cout << "x" << i + 1 << "=" << x[i][0] << endl;

//trying spectral radius calculation
double p_G = 0, sum1 = 0, sum2 = 0, base = 0, expo = 0;
for (i = 0; i < n; i++)
{
	sum1 += (x[i][0] - T_dash[i][0])*(x[i][0] - T_dash[i][0]);
}
cout << sum1 << "\n";
for (i = 0; i < n; i++)
```

```
                {
                        sum2 += x[i][0] * x[i][0];
                }
                cout << sum2 << "\n";
                base = sum1 / sum2;
                cout << " base = " << base;
                expo = 1 / l;
                cout << "expo = " << expo;
                p_G = pow(base, expo);
                cout << "\nspectral radius of G matrix = " << p_G;
                cout << "\nconvergence rate = " << -1 * log(p_G);
                cout << "\ncontinue (1 or 0)?";
                cin >> ans;
        };
        _getch();
        return 0;
}
```

Result

```
soumick@soumick-HP-Pavilion-Laptop-15-cc1xx:~/hpsc_old_codes$ g++ seidel.cpp
soumick@soumick-HP-Pavilion-Laptop-15-cc1xx:~/hpsc_old_codes$ ./a.out
[a].[x]=[b]
Enter Matrix a:
a[0,0] = 6
a[0,1] = 2
a[0,2] = 0
a[1,0] = 0
a[1,1] = 12
a[1,2] = 2
a[2,0] = 0
a[2,1] = 0
a[2,2] = 4
Enter Matrix b:
b[0][0] = 4
b[1][0] = 7
b[2][0] = 2

Iteration no: 0

Iteration no: 1

Iteration no: 2

Iteration no: 3
x1=0.5
x2=0.5
x3=0.5
```

3. **Jacobi_openmp.c**

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

int main ( );

/******************************************************************************/

int main ( )

/******************************************************************************/
/*
  Purpose:

        MAIN is the main program for JACOBI_OPENMP.

  Discussion:

        JACOBI_OPENMP carries out a Jacobi iteration with OpenMP.

  Licensing:

        This code is distributed under the GNU LGPL license.

  Modified:

        31 January 2017

  Author:

        John Burkardt
*/
{
  double *b;
  double d;
  int i;
  int it;
  int m;
  int n;
  double r;
  double t;
  double *x;
  double *xnew;
```

```
  m = 5000;
  n = 50000;

  b = ( double * ) malloc ( n * sizeof ( double ) );
  x = ( double * ) malloc ( n * sizeof ( double ) );
  xnew = ( double * ) malloc ( n * sizeof ( double ) );

  printf ( "\n" );
  printf ( "JACOBI_OPENMP:\n" );
  printf ( "  C/OpenMP version\n" );
  printf ( "  Jacobi iteration to solve A*x=b.\n" );
  printf ( "\n" );
  printf ( "  Number of variables  N = %d\n", n );
  printf ( "  Number of iterations M = %d\n", m );

  printf ( "\n" );
  printf ( "  IT    l2(dX)  l2(resid)\n" );
  printf ( "\n" );
  /*
  Each thread in the loop inside the block below will have it's own copy of i
  All other variables in the loops will be shared by default because of the type
  of abstaction on which OpenMP is buit.
  */
# pragma omp parallel private ( i )
  {
/*
  Set up the right hand side for Ax=b

  _
*/
# pragma omp for
          for ( i = 0; i < n; i++ )
          {
          b[i] = 0.0;
          }

          b[n-1] = ( double ) ( n + 1 );
/*
  Initialize the solution estimate to 0.
  Exact solution is (1,2,3,...,N).
*/
# pragma omp for
          for ( i = 0; i < n; i++ )
          {
          x[i] = 0.0;
          }
```

```
  }
/*
  Iterate M times.
  This outermost loop cannot be parallelised in a true sense since each iteration
  is dependent upon values from the previous iteration so threads corresponding
  to differnt iterations cannot run concurrently.

*/
  for ( it = 0; it < m; it++ )
  {
/*The private directive declares data to have a separate copy in the memory of each
thread.
  Such private variables are initialized as they would be in a main program.
  Any computed value goes away at the end of the parallel region
*/
# pragma omp parallel private ( i, t )
        {
/*
  Jacobi update.
  _ A matrix : The main diagonal elements are all 2.
                        The elements of the diagonals adjacent to the main diagonal
                        on either side are all -1.
                        Rest all elements are zero
  The for loop below has a cannonical shape so it can be multithreaded by the
complier.
  Each thread in the loop below modifies a different index of the array xnew[]
  so there is no requirement of snooping of any kind.
*/
# pragma omp for
        for ( i = 0; i < n; i++ )
        {
        xnew[i] = b[i];
        if ( 0 < i )
        {
        xnew[i] = xnew[i] + x[i-1];
        }
        if ( i < n - 1 )
        {
        xnew[i] = xnew[i] + x[i+1];
        }
        xnew[i] = xnew[i] / 2.0;
        }
/*
  Difference.
  d below is the reduction variable that will hold the result of the summation
```

```
  of values across threads.
*/
        d = 0.0;
# pragma omp for reduction ( + : d )
        for ( i = 0; i < n; i++ )
        {
        d = d + pow ( x[i] - xnew[i], 2 );
        }
/*
  Overwrite old solution.
  This is an "embarassingly parallel" block
*/
# pragma omp for
        for ( i = 0; i < n; i++ )
        {
        x[i] = xnew[i];
        }
/*
  Residual.
  r below is the reduction variable that will hold the result of the summation
  of values across threads. OpenMP will take care of details like storing partial
  sums in private variables and then adding the partial sums to the shared variable.
*/
        r = 0.0;
# pragma omp for reduction ( + : r )
        for ( i = 0; i < n; i++ )
        {
        t = b[i] - 2.0 * x[i];
        if ( 0 < i )
        {
        t = t + x[i-1];
        }
        if ( i < n - 1 )
        {
        t = t + x[i+1];
        }
        r = r + t * t;
        }
/*This just prints the iteration no. and L2 norm values
The omp master directive means that the section of code that follows
must be run only by the master thread
*/
# pragma omp master
        {
        if ( it < 10 || m - 10 < it )
        {
```

```
        printf ( "  %8d  %14.6g  %14.6g\n", it, sqrt ( d ), sqrt ( r ) );
        }
        if ( it == 9 )
        {
        printf ( "  Omitting intermediate results.\n" );
        }
        }

        }

  }
/*
  Write part of final estimate.
  No parallel directive below
*/
  printf ( "\n" );
  printf ( "  Part of final solution estimate:\n" );
  printf ( "\n" );
  for ( i = 0; i < 10; i++ )
  {
        printf ( "  %8d  %14.6g\n", i, x[i] );
  }
  printf ( "...\n" );
  for ( i = n - 11; i < n; i++ )
  {
        printf ( "  %8d  %14.6g\n", i, x[i] );
  }
/*
  Free memory to avoid creating orphaned blocks in memory created by dynamic
allocation.
*/
  free ( b );
  free ( x );
  free ( xnew );
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "JACOBI_OPENMP:\n" );
  printf ( "  Normal end of execution.\n" );

  return 0;
}
```

Result

```
soumick@soumick-HP-Pavilion-Laptop-15-cc1xx:~/MPI_codes$ gcc -o jacobi_openmp jacobi_openmp.c -lm
soumick@soumick-HP-Pavilion-Laptop-15-cc1xx:~/MPI_codes$ ./jacobi_openmp

JACOBI_OPENMP:
  C/OpenMP version
  Jacobi iteration to solve A*x=b.

  Number of variables  N = 50000
  Number of iterations M = 5000

  IT     l2(dX)    l2(resid)

         0         25000.5         25000.5
         1         12500.2           17678
         2         8839.01         13975.7
         3         6987.85         11692.9
         4         5846.46         10126.4
         5         5063.18         8976.06
         6         4488.03         8090.91
         7         4045.45         7385.96
         8         3692.98         6809.52
         9         3404.76         6328.22
  Omitting intermediate results.
      4991         31.6207         63.2319
      4992          31.616         63.2224
      4993         31.6112         63.2129
      4994         31.6065         63.2034
      4995         31.6017          63.194
      4996          31.597         63.1845
      4997         31.5922          63.175
      4998         31.5875         63.1655
      4999         31.5828          63.156

  Part of final solution estimate:

         0               0
         1               0
         2               0
         3               0
         4               0
         5               0
         6               0
         7               0
         8               0
         9               0
...
     49989         43819.8
     49990         44378.4
     49991           44937
     49992         45497.5
     49993         46058.1
     49994         46620.2
     49995         47182.4
     49996         47745.7
     49997         48308.9
     49998         48872.9
     49999         49436.8

JACOBI_OPENMP:
  Normal end of execution.
```

---

4. **mxm_openmp.c**

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <time.h>
# include <omp.h>
```

```
int main ( void );
void timestamp ( void );

/******************************************************************************/

int main ( void )

/******************************************************************************/
/*
  Purpose:

          MAIN is the main program for MXM_OPENMP.

  Licensing:

          This code is distributed under the GNU LGPL license.

  Modified:

          13 October 2011

  Author:

          John Burkardt
*/
/*
The code below solves matrix matrix multiplication problem using
OpenMP for parallel execution
*/
{
  double a[500][500];
  double angle;
  double b[500][500];
  double c[500][500];
  int i;
  int j;
  int k;
  int n = 500;
  double pi = 3.141592653589793;
  double s;
  int thread_num;
  double wtime;

  timestamp ( );

  printf ( "\n" );
```

```
  printf ( "MXM_OPENMP:\n" );
  printf ( "  C/OpenMP version\n" );
  printf ( "  Compute matrix product C = A * B.\n" );

  //assigning thread number to be the maximum possible during runtime
  //More threads means less iterations per thread => lower runtime
  thread_num = omp_get_max_threads ( );

  printf ( "\n" );
  //The integer returned by this function may be less than the total no. of
  //physical processors in the multiprocessor depending on how the run-tme system
  //gives access to processors
  printf ( "  The number of processors available = %d\n", omp_get_num_procs ( ) );
  printf ( "  The number of threads available   = %d\n", thread_num );

  printf ( "  The matrix order N                = %d\n", n );

  s = 1.0 / sqrt ( ( double ) ( n ) );

  //The omp_get_wtime routine returns elapsed wall clock time in seconds
  wtime = omp_get_wtime ( );

//inside this block every time the program forks all threads will have their own copy
//of variables declared private by the directive below. Rest of the shared variables
//accessible by all threads at a common memory address.
# pragma omp parallel shared ( a, b, c, n, pi, s ) private ( angle, i, j, k )
{
/*
  Loop 1: Assign value to matrix A.
*/
  /*
  Here we parallelize the outer loop and not the inner loop.
  Reason: If we parallelize the inner loop, then the program will fork
  and join threads for every iteration of the outer
  loop. The fork/join overhead may very well be greater than the time saved by
  dividing the execution of the n iterations of the inner loop among multiple threads.
  On the other hand, if we parallelize the outer loop, the program only incurs the
  fork/join overhead once
  */
  # pragma omp for
  for ( i = 0; i < n; i++ )
  {
        for ( j = 0; j < n; j++ )
        {
        angle = 2.0 * pi * i * j / ( double ) n;
        a[i][j] = s * ( sin ( angle ) + cos ( angle ) );
```

```
        }
  }
/*
  Loop 2: Copy A into B.
  Here we parallelize the outer loop and not the inner loop.
  Reason mentioned in Loop 1 comment.
*/
  # pragma omp for
  for ( i = 0; i < n; i++ )
  {
        for ( j = 0; j < n; j++ )
        {
        b[i][j] = a[i][j];
        }
  }
/*
  Loop 3: Compute C = A * B.
  Here we parallelize the outer loop and not the inner loops.
  Reason mentioned in Loop 1 comment.
*/
  /*
  Here by structure no 2 threads accesses the same index of the 3 matrices
  so no cache coherence protocols required seperately
  */
  # pragma omp for
  for ( i = 0; i < n; i++ )
  {
        //Traversing along row i of A
        for ( j = 0; j < n; j++ )
        {
        //Traversing along column j of B
        c[i][j] = 0.0;
        for ( k = 0; k < n; k++ )
        {
        //updating value of 1 element of C
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
        }
  }

}
//Calculating time taken by the code from the point A was initialised to the point
//where C was calculated and all threads joined
  wtime = omp_get_wtime ( ) - wtime;
  printf ( "  Elapsed seconds = %g\n", wtime );
  printf ( "  C(100,100)  = %g\n", c[99][99] );
```

```
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "MXM_OPENMP:\n" );
  printf ( "  Normal end of execution.\n" );

  printf ( "\n" );
  timestamp ( );

  return 0;
}
/******************************************************************************/

void timestamp ( void )

/******************************************************************************/
/*
  Purpose:

        TIMESTAMP prints the current YMDHMS date as a time stamp.

  Example:

        31 May 2001 09:45:54 AM

  Licensing:

        This code is distributed under the GNU LGPL license.

  Modified:

        24 September 2003

  Author:

        John Burkardt

  Parameters:

        None
*/
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
```

```
    const struct tm *tm;
    time_t now;

    now = time ( NULL );
    tm = localtime ( &now );

    strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );

    printf ( "%s\n", time_buffer );

    return;
  # undef TIME_SIZE
  }
```

Result

```
soumick@soumick-HP-Pavilion-Laptop-15-cc1xx:~/MPI_codes$ gcc -o mxm_openmp mxm_openmp.c -lm -fopenmp
soumick@soumick-HP-Pavilion-Laptop-15-cc1xx:~/MPI_codes$ ./mxm_openmp
15 June 2020 02:23:02 PM

MXM_OPENMP:
  C/OpenMP version
  Compute matrix product C = A * B.

  The number of processors available = 8
  The number of threads available    = 8
  The matrix order N                 = 500
  Elapsed seconds = 0.182224
  C(100,100)  = 1

MXM_OPENMP:
  Normal end of execution.
```

---

5. **Mpi_hello.c**

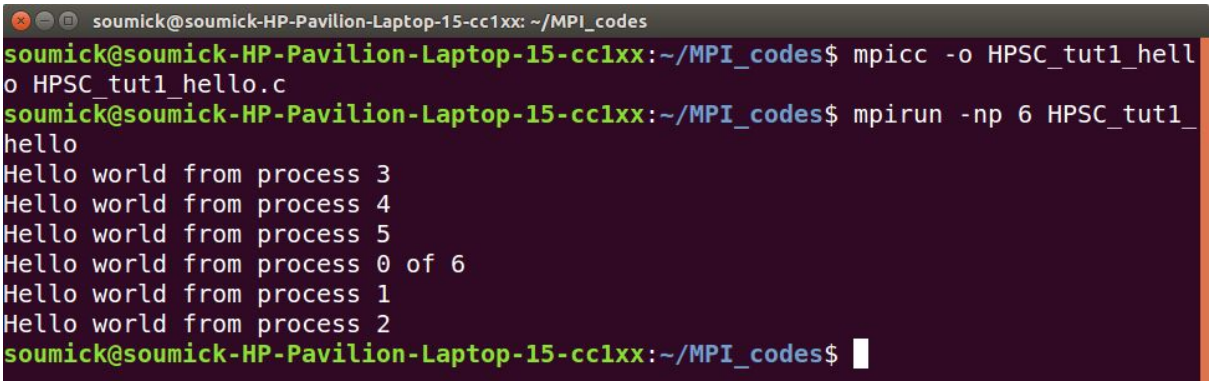
```
  //HPSC MPI tut 1
#include <stdio.h>
#include <mpi.h>


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

  MPI_Init (&argc, &argv);    /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);    /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);    /* get number of processes */
  if(rank == 0)
     printf( "Hello world from process %d of %d\n", rank, size );
  else
     printf( "Hello world from process %d\n", rank);
```

```
  MPI_Finalize();
  return 0;
}
```

Result



---

6. **mpi_send_receive.c**

```
/*
A simple MPI example program using standard mode send and receive

The program consists of two processes. Process 0 sends a large message
to the receiver. This receives the message and sends it back.
This program deadlocks if the the send and receive calls are in
the wrong order, i.e. if both proceesses first try to send, because
the message is large enough so that standard mode send does not use
buffered communication.

Compile the program with 'mpicc -O3 send-standard-large.c -o send-standard-large'
Run it on two processes.
*/

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

int main(int argc, char* argv[]) {
  const int K = 1024;
  const int msgsize = 256*K;  /* Messages will be of size 1 MB */
  int np, me, i;
  int *X, *Y;
  int tag = 42;
  MPI_Status  status;

  MPI_Init(&argc, &argv);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &me);

/* Check that we run on exactly two processes */
if (np != 2) {
        if (me == 0) {
        printf("You have to use exactly 2 processes to run this program\n");
        }
        MPI_Finalize();          /* Quit if there is only one process */
        exit(0);
}

/* Allocate memory for large message buffers */
X = (int *) malloc(msgsize*sizeof(int));
Y = (int *) malloc(msgsize*sizeof(int));


/* Initialize X and Y */
for (i=0; i<msgsize; i++) {
        X[i] = 12345;
        Y[i] = me;
}

//if (me == 0) {

        printf("Message size is %d bytes\n", msgsize*sizeof(int));

        printf("Process %d sending to process 1\n", me);
        MPI_Send(X, msgsize, MPI_INT, 1, tag, MPI_COMM_WORLD);   //send call a
        printf("Process %d receiving from process 1\n", me);
        MPI_Recv (Y, msgsize, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);  //recv call
b
        printf ("Y now has the value %d\n", Y[0]);

//} else { /* me == 1 */

        MPI_Recv (Y, msgsize, MPI_INT, 0, tag, MPI_COMM_WORLD, &status); //recv call
a
        MPI_Send (Y, msgsize, MPI_INT, 0, tag, MPI_COMM_WORLD);   //send call b

// }
        /*
        After removing the if-else condition the output was

        Message size is 1048576 bytes
        Process 0 sending to process 1
```

```
        Message size is 1048576 bytes
        Process 1 sending to process 1
        [hang/waiting]

        Explanation:
        The send and receive protocol used above is "blocking".
        Both Process 0 and Process 1 (running parallely) execute send call a
        Then Both process wait for send call b from the other processor, but before
        the receiver sends the message back it has to receive first(recv call a) but
        here recv call b is called after sender's receive call (recv call b).

        Thus we see that both process are waiting for their respective receivers
        to send back the message which will never happen because of the order conflict
        in the code.


        */

  MPI_Finalize();
  exit(0);
}
```

Result

a. ***With the if else construct***

```
soumick@soumick-HP-Pavilion-Laptop-15-cc1xx:~/MPI_codes$ mpirun -np 2 HPSC_tut1_
send_recv_original
Message size is 1048576 bytes
Process 0 sending to process 1
Process 0 receiving from process 1
Y now has the value 12345
```

b. ***Without the if else construct***

```
soumick@soumick-HP-Pavilion-Laptop-15-cc1xx:~/MPI_codes$ mpirun -np 2 HPSC_tut1_
send_recv
Message size is 1048576 bytes
Process 0 sending to process 1
Message size is 1048576 bytes
Process 1 sending to process 1
```
