(HPSC **5576** ELIZABETH JESSUP)

HIGH PERFORMANCE SCIENTIFIC COMPUTING

:: Homework / 2

:: Student / Florian Rappl

2 problems / **20** points

Problem 1

Task:

Programming assignment 3.7.1 from Pacheco's PPMPI textbook: Ring "Hello World" (p. 52), *10 pts*.

Solution:

I just took the "Greetings" sample program and made some modifications. Before I explain my code I would like to answer the questions.

Question (1):

Should the program send first and then receive, or receive and then send? Does it matter?

Answer (1):

We should send first, because we do it synchronized, i.e. we do not work with events and asynchronous requests. Therefore we are going into listening mode when receiving, which means that if all (processors) are just listening we will get stuck. On the other hand it is quite safe just to send, because all sent messages will be buffered (those are blocking sends), i.e. we will not lose any data if we keep on doing other things while we might just receive data.

Question (2):

What happens when the program is run on one processor? (If it breaks, fix it!)

Answer (2):

My program broke with the error message:

```
[0] fatal error
Fatal error in PMPI_Send: Other MPI error, error stack:
PMPI_Send(150): MPI_Send(buf=0x0020F924, count=100, MPI_CHAR, dest=0,
tag=0, MPI_COMM_WORLD) failed
PMPI_Send(125): DEADLOCK: attempting to send a message to the local process
without a prior matching receive
```

The problem is indeed the MPI_Send which (apparently) does not work in a loop (sending from one node to the same node). The simple fix was to distinguish between $p > 1$ and $p == 1$, where p is the number of processors.

The output with p=1:

```
Greetings from process 0!
```

The output with p=2:

```
Greetings from process 1!
Greetings from process 0!
```

The output with p=32:

```
Greetings from process 31!
Greetings from process 30!
Greetings from process 29!
Greetings from process 28!
Greetings from process 27!
Greetings from process 26!
Greetings from process 25!
Greetings from process 24!
Greetings from process 23!
Greetings from process 22!
Greetings from process 21!
Greetings from process 20!
Greetings from process 19!
Greetings from process 18!
Greetings from process 17!
Greetings from process 16!
Greetings from process 15!
Greetings from process 14!
Greetings from process 13!
Greetings from process 12!
Greetings from process 11!
Greetings from process 10!
Greetings from process 9!
Greetings from process 8!
Greetings from process 7!
Greetings from process 6!
Greetings from process 5!
Greetings from process 4!
Greetings from process 3!
Greetings from process 2!
Greetings from process 1!
Greetings from process 0!
```

About the code:

First of all the greetings message is generated on every node. After that generation we distinguish between the two cases. In the $p == 1$ case we just print the generated message. In the other case $p > 1$ we send the message to the next node determined by $my\_rank\ +\ 1 \,\%\, p$, i.e. always to $my\_rank\ +\ 1$ except if the rank is $p - 1$ (the last node) – this one will send to process 0.

This starts our circle of messages to process 0 (the root process). While process 0 has now p messages to receive (all $p - 1$ messages of the other processors plus the one which was originally sent by it), process 1 has only 1 message to receive, process 2 has 2 messages and so on.

In the end while process 0 just prints out all received messages, all the other nodes are just sending them again using the same determinism as before.

Code printout:

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

main(int argc, char* argv[]) {
    int         my_rank;        /* rank of process      */
    int         p;              /* number of processes  */
    int         source;         /* rank of sender       */
```

```
    int          dest;          /* rank of receiver     */
    int          tag = 0;       /* tag for messages     */
    char         message[100];  /* storage for message  */
    char         my_name[64];   /* name of machine      */
    int          my_name_len;   /* length of my_name    */
    MPI_Status   status;        /* return status for    */
                                /* receive              */
      int          length = 1;

    /* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank  */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    /* Modified from Pacheco -- print machine name */
    MPI_Get_processor_name( my_name, &my_name_len );

      sprintf(message, "Greetings from process %d!",
            my_rank);

    /* Create message */
      if(p > 1)
      {
            MPI_Send(message, 100, MPI_CHAR, ( my_rank + 1 ) % p,
                  tag, MPI_COMM_WORLD);

            if(my_rank == 0)
                  length = p;
            else
                  length = my_rank;

            for (source = 0; source < length; source++)
            {
                  MPI_Recv(message, 100, MPI_CHAR, (my_rank-1+p) % p, tag,
                        MPI_COMM_WORLD, &status);

                  if(my_rank == 0)
                        printf("%s\n", message);
                  else
                        MPI_Send(message, 100, MPI_CHAR, ( my_rank + 1 ) %
p,
                              tag, MPI_COMM_WORLD);
            }
      }
      else
            printf("%s\n", message);

    /* Shut down MPI */
    MPI_Finalize();
} /* main */
```

Problem 2

Task:

Programming assignment 4.7.2 from Pacheco's PPMPI textbook: Simpson's Rule (p. 64), *10 pts*.

Solution:

I decided to distribute all the function values over the processors. Since we have 3 different function values for each segment (Simpson's rule), we have 1 shared function value at the boundaries (beginning and end). The middle one with weighting 4 is always a single one. Therefore we have to weigh the shared ones with 2 instead of 1. Overall in an n segment integration using Simpson's rule we have $2n + 1$ function values. Therefore the best processor count is obviously $2n + 1$ – everything above it is redundant and everything below it results either in idle time for some processors at the final stage (with $2n$ processors being worst case) or a longer computation time (worst case here is a single processor – it has to do $2n + 1$ function value evaluations plus the whole summation). Since a parabola is exact I've decided to pick $f(x) = x^2$ as function. The function as well as the interval is hardcoded – even though it would not require much skill to determine the interval over a command line argument (since this has been coded with the number of segments n – it is just a simple enhancement).

The output with p=1 and n=10000:

```
Precision:      n = 10000
Cores:          p = 1
Area:           A = 21333.000000
Lower Bound:    a = 1.000000
Upper Bound:    b = 40.000000
```

About the command line parser:

Since it has been a while working in **pure C** I do not know if my solution there is the best or if it could be somehow improved (there is always room for improvement). I am just using the number of arguments *argc* and go through that array of strings (*char**) starting with the first argument (since *argv[0]* is just the name of the program – reflected).

Next I am just using the in the *string.h* inbuilt function to compare two strings and have a closer look if I actually get a match for "**-n**" (the number of intervals). For "**-v**" I activate the verbose mode and for "**-?**" I am printing a short help for the user concerning the usage of the command line arguments. The closer look for the number of intervals is just looking if there is a next argument (as there should be one – a number) and if so – whether it really is a number.

If it is not a number / or a wrong number (below 1) I give out an error and reset n to my standard number of intervals (I actually did use that number twice in the code – could have been done more elegant using a macro – but I wanted to preserve the readability of the code). The interval could have been implemented using the "**-n**" code for "**-a**" and "**-b**" as well.

The output with p=16, activated verbose option and n=10:

```
P: coe. loop    x               f(x)
0: 1    0       1.000000        1.000000
0: 2    16      32.200000       1036.840000
1: 4    1       2.950000        8.702500
2: 2    2       4.900000        24.010000
3: 4    3       6.850000        46.922500
4: 2    4       8.800000        77.440000
5: 4    5       10.750000       115.562500
6: 2    6       12.700000       161.290000
7: 4    7       14.650000       214.622500
8: 2    8       16.600000       275.560000
9: 4    9       18.550000       344.102500
10: 2   10      20.500000       420.250000
11: 4   11      22.450000       504.002500
12: 2   12      24.400000       595.360000
13: 4   13      26.350000       694.322500
14: 2   14      28.300000       800.890000
15: 4   15      30.250000       915.062500
1: 4    17      34.150000       1166.222500
2: 2    18      36.100000       1303.210000
3: 4    19      38.050000       1447.802500
4: 1    20      40.000000       1600.000000
Precision:      n = 10
Cores:          p = 16
Area:           A = 21333.000000
Lower Bound:    a = 1.000000
Upper Bound:    b = 40.000000
```

Why the implementation is correct:

Most of the things in the code are pretty "standard", i.e. the do not need to be explained. The implementation for the verbose option may be a little bit strange but since this option is just to be a debugging helper / information giver there was no need in implementing this function very fancy or effective. The most interesting part is the calculation of the function values. The summation of the subareas would be more effective in a tree (as I recommended in the lecture) but since this was not part of the task I just ignored this issue. If we take for example an integration using Simpson's rule with $n = 4$ we obtain

coeff.	1	4	2	4	2	4	2	4	1
j*h	0	$\frac{1}{2}$	1	$\frac{3}{2}$	2	$\frac{5}{2}$	3	$\frac{7}{2}$	4
loop	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

This makes $2n + 1$ values and the coefficient distribution I explained in the first paragraph. Therefore we can see that for our loop we need to go from 0 to $2n$ – including both values (therefore $2n + 1$ iterations). Also we see that we can determine the coefficient by using if and a modulo operation. If the loop is at 0 or $2n$ we set the coefficient to 1 – else we determine the coefficient using $2 + (i \,\%\, 2) \cdot 2$, which gives us 2 for even numbers and 4 for odd numbers.

At last we always add the half of h to our local a (which has been set in the beginning properly) times the number of processes running. The h in the program is already $h/2$ to reduce the number of divisions needed. So overall I am splitting up the big function evaluation loop (from 0 to $2n$), so that each process begins at his number and goes in steps which are equal to the number of processes.

Code printout:

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

double fx(double x);            /* prototype of f(x)      */

int main(int argc, char* argv[])
{
      /* Standard header */
    int         my_rank;        /* rank of process        */
    int         p;              /* number of processes    */
    int         source;         /* rank of sender         */
    int         dest;           /* rank of receiver       */
    int         tag = 0;        /* tag for messages       */
    char        message[100];   /* storage for message    */
    char        my_name[64];    /* name of machine        */
    int         my_name_len;    /* length of my_name      */
    MPI_Status  status;         /* return status for recv */

      /* Simpson rule specific */
      int n = 10000;            /* precision       */
      int i;                    /* some loop       */
      double a = 1.0;           /* starting point  */
      double b = 40.0;          /* final point     */
      double h;                       /* spacing         */
      double total = 0.0;       /* total subarea   */
      double area = 0.0;        /* total area      */
      double loc_a;             /* local starting  */
      int verbose = 0;          /* verbose bool    */
      double fvalue;            /* stored f(x)     */

      /* Command line args parser */
      for(i = 1; i < argc; i++)
      {
            /* if we have the -n */
            if(strcmp(argv[i],"-n") == 0)
            {
                  /* but nothing else specified */
                  if(i == argc - 1)
                  {
                        printf("A number of intervals must be specified
using -n.\n");
                        break;
                  }
                  /* or we probably have a number */
                  n = atoi(argv[++i]);
                  if(n < 1)
                  {
                        /* but that is not a valid number */
```

```
                        printf("Wrong input for n. Must be greater than 
0.\n");
                        n = 10000;
                  }
            }
            /* if we have the verbose statement */
            else if(strcmp(argv[i],"-v") == 0)
                  verbose = 1;
            /* if we have some help statement */
            else if(strcmp(argv[i],"-?") == 0)
            {
                  printf("Command line arguments\n");
                  printf("======================\n");
                  printf("-n X\t sets interval to X\n");
                  printf("-v\t verbose mode\n");
                  printf("-?\t displays this help\n");
                  printf("======================");
                  return;
            }
      }

      /* Calculate spacing  */
      h = (b-a)/(double)n/2.0;

    /* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank  */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

      /* Begin calculating */
      loc_a = a + (double)my_rank * h;

      if(verbose == 1 && my_rank == 0)
            printf("P: coe.\tloop\tx\t\tf(x)\n");

      /* Best performance for n+1 % p == 0 else some */
      /* idle time for n+1 % p processors            */
      for(i = my_rank; i <= 2*n; i+=p)
      {
            if(i == 2*n || i == 0)
                  dest = 1;
            else
                  dest = 2 + (i % 2) * 2;

            fvalue = fx(loc_a);

            /* add new "area parts to the total (sub)area */
            total += (double)dest * fvalue * h / 3.0;

            if(verbose == 1)
            {
                  sprintf(&message, "%d: %d\t%d\t%f\t%f",
                        my_rank, dest, i, loc_a, fvalue);
                  if(my_rank == 0)
                        printf("%s\n", message);
                  else
                        MPI_Send(&message, 100, MPI_CHAR, 0, 0,
```

```
                                    MPI_COMM_WORLD);
                }

                /* just to save some multiplication and addition */
                if(i + p <= 2 * n)
                        loc_a += (double)p * h;
        }

        if(verbose == 1 && my_rank == 0)
        {
                for(i = 1; i <= 2 * n; i++)
                        if(i % p != 0)
                        {
                                MPI_Recv(&message, 100, MPI_CHAR, i%p, 0,
                                                MPI_COMM_WORLD, &status);
                                printf("%s\n", message);
                        }
        }

        /* Collect all the (sub)areas or send the (sub)area */
        if(my_rank == 0)
        {
                /* Adding the own subarea to the total one */
                area += total;

                /* Doing a manual Reduce */
                for(i = 1; i < p; i++)
                {
                        MPI_Recv(&total, 1, MPI_2DOUBLE_PRECISION, i, 0,
                                MPI_COMM_WORLD, &status);
                        area += total;
                }

                /* Print outcome */
                printf("Precision:\tn = %d\n", n);
                printf("Cores:\t\tp = %d\n", p);
                printf("Area:\t\tA = %f\n", area);
                printf("Lower Bound:\ta = %f\n", a);
                printf("Upper Bound:\tb = %f\n", b);
        }
        else
                MPI_Send(&total, 1, MPI_2DOUBLE_PRECISION, 0, 0,
MPI_COMM_WORLD);

      /* Shut down MPI */
      MPI_Finalize();

        return 0;
} /* main */

double fx(double x)
{
        return x*x;
} /* function */
```