Project Assignment 1

—

MPI: Message-Passing Interface

Multiprocessor Systems, DV2544

Håkan Grahn
Blekinge Institute of Technology

March 31, 2017

1 Introduction

The task in this project assignment is to implement two parallel algorithms using the Message-Passing Interface (MPI) [1, 3, 4].

The examination of this projekt is done by sending in the project, with complete source code and a report in PDF format, before the examination deadline. The examination deadline is April 23, 2017 at 23:59. The reports and source code should be submitted on the course page at It's Learning.

2 Goals

This project assignment serves two purposes:

- Introduce you to basic MPI programming.
- Give some experiece of how data partitioning and inter-node communication impact the performance of a parallel application.

The rationale behind these goals are that MPI programming is often used for high-performance computing on clustered systems, where communication overhead and data distribution are of primary concern when it comes to performance.

3 Preconditions

3.1 Prerequisites

- You are supposed to have good programming experience and not be new to C programming.
- You are supposed to have basic knowledge about working in a Unix environment.

- Operating systems and real-time issues and concepts should not be unfamiliar to you.

3.2 Laboratory groups

You are encouraged to work in groups of two. Groups larger than two is *not* accepted.

Discussion and help between laboratory groups are encouraged. It is normally not a problem, *but* watch out so that you do not cross the border to cheating, see section 3.5 below.

3.3 Lecture support

There is one lecture on MPI-programming in general, e.g., introducing the general concepts of message-passing programming and an overview of the MPI functions. In addition to this, you are expected to search for additional information on your own.

3.4 Examination

See section 4.3.

3.5 Cheating

All work that is not your own should be properly referenced. If not, it will be considered as cheating and reported as such to the university disciplinary board.

4 Project Tasks to Complete

In this project assignment you are going to implement parallel MPI versions of two different algorithms:

1. Blockwise (2-dimensional data partitioning, checker-board) Matrix-Matrix multiplication (described in Section 4.1)

2. Laplace approximation using SOR (described in Section 4.2)

The algorithms shall be implemented i C using MPI, and compile and execute correctly on our lab server (kraken.tek.bth.se).

4.1 MPI-based Blocked Matrix-Matrix Multiplication

4.1.1 Problem Description

The sequential algorithm for performing Matrix-Matrix multiplication is found in Chapter 8 (Algorithm 8.2) in [2], see Figure 1. The sequential version of matrix-matrix multiplication written in C is found in Appendix A.

An exemple of an implementation of *blockwise rowwise* 1-dimensional Matrix-Matrix multiplication using MPI is found in Appendix B. You are allowed to use the code as basis your own *blockwise 2-dimensional data partitioned* (checker-board) implementation.

```
1.   procedure MAT_MULT (A, B, C)
2.   begin
3.       for i := 0 to n − 1 do
4.           for j := 0 to n − 1 do
5.               begin
6.                   C[i, j] := 0;
7.                   for k := 0 to n − 1 do
8.                       C[i, j] := C[i, j] + A[i, k] × B[k, j];
9.               endfor;
10.  end MAT_MULT
```

Algorithm 8.2 The conventional serial algorithm for multiplication of two $n \times n$ matrices.

Figure 1: Algorithm for matrix-matrix multiplication.

4.1.2 Tasks to Complete

You are supposed to do the following:

- Write a parallel implementation of *blockwise 2-dimensional data partitioned* (checkerboard) Matrix-Matrix multiplication using MPI.
- The parallel implementation shall execute correctly on 1, 2, 4, and 8 MPI processes (nodes).
- Compare the performance (i.e., execution time) of your blocked version with the performance the parallel version given to you. The measurements should be done on 1, 2, 4, and 8 MPI processes (nodes). Calculate the speedup for 2, 4, and 8 MPI processes.

The source code for both the sequential version and the row-wise MPI-implementation of Matrix-Matrix multiplication are found on the course home page at It's Learning.

4.2 MPI-based LaPlace Approximation

4.2.1 Problem Description

The problem description is based on a two-dimentional grid of data values. The values on the borders are kept constant, and rest of the elements will be set to the average of their neighbors.

$$v_{x,y} = \frac{v_{i-1,y-1}+v_{i-1,y+1}+v_{i+1,y-1}+v_{i+1,y+1}}{4}$$

This task might seem like a waste of time, but it does something useful. When the interior values are set to their neighbors' average, they satisfy an approximation of the two-dimensional LaPlace equation:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

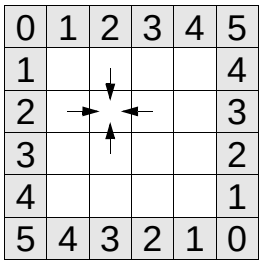

Figure 2: LaPlace approximation.

where x and y represent coordinates in space, and f is the function, i.e., the values that we are computing.

To help you with the task, we provide a sequential version of LaPlace approximation of a matrix using the Successive Over-Relaxation (SOR) method. Successive Over-Relaxation (SOR) is a numerical method used to speed up convergence of the Gauss-Seidel method. A similar method can be used for any slowly converging iterative process. For more information about the algorithm, look at the project introduction slides provided. The source code for a sequential implementation of SOR is found in Appendix C.

4.2.2 Tasks to Complete

You are supposed to do the following:

- Write a parallel MPI implementation of Laplace approximation using SOR.
- Measure the speedup of your parallel version on 2, 4, and 8 MPI processes (nodes). The measurements should be done on a matrix of size 2048x2048 elements.

4.3 Examination

Prepare and submit a `tar`-file (or `zip`-file) containing:

- **Source code:** The source-code for working solutions to the tasks in sections 4.1 and 4.2, i.e., a listing of your *well-structured* and *well-commented* source code for your parallel version of the applications.
- Corresponding `Makefile`s, or a text-file describing how to compile and execute the project tasks.
- **Written report:** You should write a short report (approximately 2-3 pages) describing your implementation and measurements (results), as outline below. The format of the report must be pdf.
  - **Implementation:** A description of your parallel implementations of the tasks in sections 4.1 and 4.2, i.e., you should describe how

you have partitioned the work between several nodes, how the data structures are organized, etc.

- **Measurements:** You should provide *execution times* and *speedup* numbers of for five cases: the sequential version of the application, and the parallel version of the application running on 1, 2, 4, and 8 nodes (MPI processes).

All material (except the code given to you in this assignment) must be produced by the laboratory group alone.

The examiner may contact you within a week, if he/she needs some oral clarifications on your code or report. In this case, all group members must be present at that oral occasion.

References

[1] MPI Forum, *Message passing interface forum*, `http://www.mpi-forum.org`.

[2] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to parallel computing, 2nd edition*, Addison-Wesley, 2003.

[3] M Snir, *Mpi: The complete reference: Core v. 1 (scientific & engineering computation)*, MIT Press, 1998.

[4] Marc Snir, William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, and William Saphir, *Mpi: The complete reference: The mpi-2 extensions vol 2 (scientific & engineering computation)*, The MIT Press, 1998.

A Sequential matrix-matrix multiplication

```
/*****************************************************************************
 *
 * Sequential version of Matrix-Matrix multiplication
 *
 *****************************************************************************/

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

#define SIZE 1024

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];

static void
init_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++) {
            /* Simple initialization, which enables us to easy check
             * the correct answer. Given SIZE size of the matrices, then
             * the output should be
             * SIZE ... 2*SIZE ...
             * ...
             * 2*SIZE ... 4*SIZE ...
             * ...
             */
            a[i][j] = 1.0;
            if (i >= SIZE/2) a[i][j] = 2.0;
            b[i][j] = 1.0;
            if (j >= SIZE/2) b[i][j] = 2.0;
        }
}

static void
matmul_seq()
{
    int i, j, k;

    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++) {
            c[i][j] = 0.0;
            for (k = 0; k < SIZE; k++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
}

static void
print_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++)
            printf(" %7.2f", c[i][j]);
        printf("\n");
    }
```

```
}

int
main(int argc, char **argv)
{
    init_matrix();
    matmul_seq();
    //print_matrix();
}
```

B Row-wise 1-dimensional matrix-matrix multiplication in MPI

```
/**********************************************************************
 *
 * MPI-version of row-wise Matrix-Matrix multiplication
 *
 * File : matmul_mpi.c
 * Author(s) : Håkan Grahn
 * Created : 2009-01-30
 * Last Modified : 2009-01-30
 * Last Modified by : Håkan Grahn
 *
 * (c) 2009-2017 by Håkan Grahn, Blekinge Institute of Technology.
 * All Rights Reserved
 **********************************************************************/

/*
 * Compile with:
 * mpicc -o mm matmul_mpi.c
 */

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

#define SIZE 1024 /* assumption: SIZE a multiple of number of nodes */
        /* SIZE should be 1024 in our measurements in the assignment */
        /* Hint: use small sizes when testing, e.g., SIZE 8 */
#define FROM_MASTER 1 /* setting a message type */
#define FROM_WORKER 2 /* setting a message type */
#define DEBUG 0  /* 1 = debug on, 0 = debug off */

MPI_Status status;

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];

static void
init_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++) {
            /* Simple initialization, which enables us to easy check
             * the correct answer. Given SIZE size of the matrices, then
             * the output should be
             * SIZE ... 2*SIZE ...
             * ...
             * 2*SIZE ... 4*SIZE ...
             * ...
             */
            a[i][j] = 1.0;
            if (i >= SIZE/2) a[i][j] = 2.0;
            b[i][j] = 1.0;
            if (j >= SIZE/2) b[i][j] = 2.0;
        }
}
```

```
static void
print_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++)
            printf(" %7.2f", c[i][j]);
        printf("\n");
    }
}

int
main(int argc, char **argv)
{
    int myrank, nproc;
    int rows; /* amount of work per node (rows per worker) */
    int mtype; /* message type: send/recv between master and workers */
    int dest, src, offset;
    double start_time, end_time;
    int i, j, k;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0) { /* Master task */
        /* Initialization */
        printf("SIZE = %d, number of nodes = %d\n", SIZE, nproc);
        init_matrix();
        start_time = MPI_Wtime();

        /* Send part of matrix a and the whole matrix b to workers */
        rows = SIZE / nproc;
        mtype = FROM_MASTER;
        offset = rows;
        for (dest = 1; dest < nproc; dest++) {
            if (DEBUG)
                printf(" sending %d rows to task %d\n",rows,dest);
            MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            MPI_Send(&a[offset][0], rows*SIZE, MPI_DOUBLE, dest, mtype,
                MPI_COMM_WORLD);
            MPI_Send(&b, SIZE*SIZE, MPI_DOUBLE, dest, mtype,
                MPI_COMM_WORLD);
            offset += rows;
        }

        /* let master do its part of the work */
        for (i = 0; i < rows; i++) {
            for (j = 0; j < SIZE; j++) {
                c[i][j] = 0.0;
                for (k = 0; k < SIZE; k++)
                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }

        /* collect the results from all the workers */
        mtype = FROM_WORKER;
        for (src = 1; src < nproc; src++) {
            MPI_Recv(&offset, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
                &status);
```

```
            MPI_Recv(&rows, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
                &status);
            MPI_Recv(&c[offset][0], rows*SIZE, MPI_DOUBLE, src, mtype,
                MPI_COMM_WORLD, &status);
            if (DEBUG)
                printf(" recvd %d rows from task %d, offset = %d\n", rows, src, offset);
        }

        end_time = MPI_Wtime();
        if (DEBUG)
            /* Prints the resulting matrix c */
            print_matrix();
        printf("Execution time on %2d nodes: %f\n", nproc, end_time-start_time);

    } else { /* Worker tasks */
        /* Receive data from master */
        mtype = FROM_MASTER;
        MPI_Recv(&offset, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&a[offset][0], rows*SIZE, MPI_DOUBLE, 0, mtype,
            MPI_COMM_WORLD, &status);
        MPI_Recv(&b, SIZE*SIZE, MPI_DOUBLE, 0, mtype, MPI_COMM_WORLD,
            &status);
        if (DEBUG)
            printf ("Rank=%d, offset=%d, row =%d, a[offset][0]=%f, b[0][0]=%f\n",
                myrank, offset, rows, a[offset][0], b[0][0]);

        /* do the workers part of the calculation */
        for (i=offset; i<offset+rows; i++)
            for (j=0; j<SIZE; j++) {
                c[i][j] = 0.0;
                for (k=0; k<SIZE; k++)
                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        if (DEBUG)
            printf("Rank=%d, offset=%d, row =%d, c[offset][0]=%e\n",
                myrank, offset, rows, a[offset][0]);

        /* send the results to the master */
        mtype = FROM_WORKER;
        MPI_Send(&offset, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
        MPI_Send(&c[offset][0], rows*SIZE, MPI_DOUBLE, 0, mtype,
            MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

C Sequential Successive Over-Relaxation (SOR)

```
/*****************************************************
 *
 * S O R algorithm
 * ("Red−Black" solution to LaPlace approximation)
 *
 * sequential version
 *
 *****************************************************/

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

#define MAX_SIZE 4096
#define EVEN_TURN 0 /* shall we calculate the 'red' or the 'black' elements */
#define ODD_TURN 1

typedef double matrix[MAX_SIZE+2][MAX_SIZE+2]; /* (+2) − boundary elements */

volatile struct globmem {
    int  N;  /* matrix size  */
    int  maxnum;  /* max number of element*/
    char *Init;  /* matrix init type */
    double difflimit; /* stop condition */
    double w;  /* relaxation factor */
    int  PRINT;  /* print switch  */
    matrix A;  /* matrix A  */
} *glob;

/* forward declarations */
int work();
void Init_Matrix();
void Print_Matrix();
void Init_Default();
int Read_Options(int, char **);

int
main(int argc, char **argv)
{
    int i, timestart, timeend, iter;

    glob = (struct globmem *) malloc(sizeof(struct globmem));

    Init_Default();  /* Init default values */
    Read_Options(argc,argv); /* Read arguments */
    Init_Matrix();  /* Init the matrix */
    iter = work();
    if (glob->PRINT == 1)
      Print_Matrix();
    printf("\nNumber of iterations = %d\n", iter);
}

int
work()
{
    double prevmax_even, prevmax_odd, maxi, sum, w;
    int m, n, N, i;
    int finished = 0;
    int turn = EVEN_TURN;
```

```
int iteration = 0;

prevmax_even = 0.0;
prevmax_odd = 0.0;
N = glob->N;
w = glob->w;

while (!finished) {
  iteration++;
  if (turn == EVEN_TURN) {
      /* CALCULATE part A - even elements */
      for (m = 1; m < N+1; m++) {
        for (n = 1; n < N+1; n++) {
            if (((m + n) % 2) == 0)
              glob->A[m][n] = (1 - w) * glob->A[m][n]
                  + w * (glob->A[m-1][n] + glob->A[m+1][n]
                      + glob->A[m][n-1] + glob->A[m][n+1]) / 4;
        }
      }
      /* Calculate the maximum sum of the elements */
      maxi = -999999.0;
      for (m = 1; m < N+1; m++) {
        sum = 0.0;
        for (n = 1; n < N+1; n++)
            sum += glob->A[m][n];
        if (sum > maxi)
            maxi = sum;
      }
      /* Compare the sum with the prev sum, i.e., check wether
       * we are finished or not. */
      if (fabs(maxi - prevmax_even) <= glob->difflimit)
        finished = 1;
      if ((iteration%100) == 0)
        printf("Iteration: %d, maxi = %f, prevmax_even = %f\n",
               iteration, maxi, prevmax_even);
      prevmax_even = maxi;
      turn = ODD_TURN;

  } else if (turn == ODD_TURN) {
      /* CALCULATE part B - odd elements*/
      for (m = 1; m < N+1; m++) {
        for (n = 1; n < N+1; n++) {
            if (((m + n) % 2) == 1)
              glob->A[m][n] = (1 - w) * glob->A[m][n]
                  + w * (glob->A[m-1][n] + glob->A[m+1][n]
                      + glob->A[m][n-1] + glob->A[m][n+1]) / 4;
        }
      }
      /* Calculate the maximum sum of the elements */
      maxi = -999999.0;
      for (m = 1; m < N+1; m++) {
        sum = 0.0;
        for (n = 1; n < N+1; n++)
            sum += glob->A[m][n];
        if (sum > maxi)
            maxi = sum;
      }
      /* Compare the sum with the prev sum, i.e., check wether
       * we are finished or not. */
      if (fabs(maxi - prevmax_odd) <= glob->difflimit)
        finished = 1;
      if ((iteration%100) == 0)
```

```
            printf("Iteration: %d, maxi = %f, prevmax_odd = %f\n",
                    iteration, maxi, prevmax_odd);
            prevmax_odd = maxi;
            turn = EVEN_TURN;
        } else {
            /* something is very wrong... */
            printf("PANIC: Something is really wrong!!!\n");
            exit(-1);
        }
        if (iteration > 100000) {
            /* exit if we don't converge fast enough */
            printf("Max number of iterations reached! Exit!\n");
            finished = 1;
        }
    }
    return iteration;
}

/*------------------------------------------------------------------------*/

void
Init_Matrix()
{
    int i, j, N, dmmy;

    N = glob->N;
    printf("\nsize = %dx%d ",N,N);
    printf("\nmaxnum = %d \n",glob->maxnum);
    printf("difflimit = %.7lf \n",glob->difflimit);
    printf("Init = %s \n",glob->Init);
    printf("w = %f \n\n",glob->w);
    printf("Initializing matrix...");

    /* Initialize all grid elements, including the boundary */
    for (i = 0; i < glob->N+2; i++) {
      for (j = 0; j < glob->N+2; j++) {
          glob->A[i][j] = 0.0;
      }
    }
    if (strcmp(glob->Init,"count") == 0) {
      for (i = 1; i < N+1; i++){
          for (j = 1; j < N+1; j++) {
            glob->A[i][j] = (double)i/2;
          }
      }
    }
    if (strcmp(glob->Init,"rand") == 0) {
      for (i = 1; i < N+1; i++){
          for (j = 1; j < N+1; j++) {
            glob->A[i][j] = (rand() % glob->maxnum) + 1.0;
          }
      }
    }
    if (strcmp(glob->Init,"fast") == 0) {
      for (i = 1; i < N+1; i++){
          dmmy++;
          for (j = 1; j < N+1; j++) {
            dmmy++;
            if ((dmmy%2) == 0)
                glob->A[i][j] = 1.0;
            else
                glob->A[i][j] = 5.0;
```

```
            }
        }
    }

    /* Set the border to the same values as the outermost rows/columns */
    /* fix the corners */
    glob->A[0][0] = glob->A[1][1];
    glob->A[0][N+1] = glob->A[1][N];
    glob->A[N+1][0] = glob->A[N][1];
    glob->A[N+1][N+1] = glob->A[N][N];
    /* fix the top and bottom rows */
    for (i = 1; i < N+1; i++) {
      glob->A[0][i] = glob->A[1][i];
      glob->A[N+1][i] = glob->A[N][i];
    }
    /* fix the left and right columns */
    for (i = 1; i < N+1; i++) {
      glob->A[i][0] = glob->A[i][1];
      glob->A[i][N+1] = glob->A[i][N];
    }

    printf("done \n\n");
    if (glob->PRINT == 1)
      Print_Matrix();
}

void
Print_Matrix()
{
    int i, j, N;

    N = glob->N;
    for (i=0; i<N+2 ;i++){
      for (j=0; j<N+2 ;j++){
          printf(" %f",glob->A[i][j]);
      }
      printf("\n");
    }
    printf("\n\n");
}

void
Init_Default()
{
    glob->N = 2048;
    glob->difflimit = 0.00001*glob->N;
    glob->Init = "rand";
    glob->maxnum = 15.0;
    glob->w = 0.5;
    glob->PRINT = 0;
}

int
Read_Options(int argc, char **argv)
{
    char *prog;

    prog = *argv;
    while (++argv, --argc > 0)
      if (**argv == '-')
          switch ( *++*argv ) {
          case 'n':
```

```
        --argc;
        glob->N = atoi(*++argv);
        glob->difflimit = 0.00001*glob->N;
        break;
      case 'h':
        printf("\nHELP: try sor -u \n\n");
        exit(0);
        break;
      case 'u':
        printf("\nUsage: sor [-n problemsize]\n");
        printf(" [-d difflimit] 0.1-0.000001 \n");
        printf(" [-D] show default values \n");
        printf(" [-h] help \n");
        printf(" [-I init_type] fast/rand/count \n");
        printf(" [-m maxnum] max random no \n");
        printf(" [-P print_switch] 0/1 \n");
        printf(" [-w relaxation_factor] 1.0-0.1 \n\n");
        exit(0);
        break;
      case 'D':
        printf("\nDefault: n = %d ", glob->N);
        printf("\n difflimit = 0.0001 ");
        printf("\n Init = rand" );
        printf("\n maxnum = 5 ");
        printf("\n w = 0.5 \n");
        printf("\n P = 0 \n\n");
        exit(0);
        break;
      case 'I':
        --argc;
        glob->Init = *++argv;
        break;
      case 'm':
        --argc;
        glob->maxnum = atoi(*++argv);
        break;
      case 'd':
        --argc;
        glob->difflimit = atof(*++argv);
        break;
      case 'w':
        --argc;
        glob->w = atof(*++argv);
        break;
      case 'P':
        --argc;
        glob->PRINT = atoi(*++argv);
        break;
      default:
        printf("%s: ignored option: -%s\n", prog, *argv);
        printf("HELP: try %s -u \n\n", prog);
        break;
      }
}
```