Università della Svizzera italiana

Institute of Computational Science ICS

High-Performance Computing Lab **Fall 2020**

Student: Gabriele Berra Discussed with: -

Solution for Project 6

Due date: 27.11.2020 23:59 (midnight)

HPC Lab 2020 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to Icorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

1. Ring maximum using MPI [10 Points]

In this first exercise, I implemented the function *max_ring.c*, parallelized with MPI. Similarly to what we did in class, I firstly defined the left and right neighbour and the value of the function

$$\max = 3 * \text{process\_rank}_i \mod (2 * \text{communicator\_size}) \tag{1}$$

for each processor. Since we have to run the function with 4 processors, we know that the global maximum of Eq. 1 is reached in the processor 2 with max = 6. In the main body of the function, on one hand, each processor, with the MPI routine `MPI_Send`, exchanges its own maximum with the next processor. On the other hand, with the MPI routine `MPI_Recv`, the processor that receives the data checks if its own maximum is bigger or lower than the maximum that it has received and keeps the bigger one. Repeating this routine in a circular manner as many times as the number of processors (i.e., 4 times in this example) gives the following results:

```
Process 0:          Max = 6
Process 2:          Max = 6
Process 3:          Max = 6
Process 1:          Max = 6
```

Of course, every time we run the script the order of the processes will change. In what follows I reported the main body of the function *max_ring.c*.

```
for (int i = 0, snd_buf = max, rcv_buf; i < size; i++, snd_buf =
    max)
{
    MPI_Request request;
    ierr = MPI_Issend(&snd_buf, 1, MPI_INT, right, 0,
       MPI_COMM_WORLD, &request);

    MPI_Status status;
    ierr = MPI_Recv(&rcv_buf, 1, MPI_INT, left, 0, MPI_COMM_WORLD,
       &status);

    if (ierr != MPI_SUCCESS){
        printf("ProcID %i did not successfully receive a value! \n
           ", my_rank);
    }

    if(max < rcv_buf){
        max = rcv_buf;
    }

}
```

2. Ghost cells exchange between neighboring processes [20 Points]

For this second exercise we have to parallelize with MPI the exchange of "ghost cells" between adjacent processors in a defined grid. The input data on which we have to work is a $(6+2)\times(6+2)$ matrix associated with each processors, in which each entry has value equal to the rank of the processor. The first step is to create a Cartesian communicator of dimension 4×4 (which is a sort of matrix for the processors) with periodic boundaries: this means that the processor 0 - which is in position $(0,0)$ - can communicate with processor 3 and processor 12 on the left and top respectively. In order to find - for each processor - the top, bottom, left and right neighbours, we have to use the MPI routine `MPI_Cart_shift` which, given a shift direction and the amount of "cells" to be shifted, returns the source and destination rank. In our case, to find the left and right neighbour we call the function as follows:

```
MPI_Cart_shift(comm_cart,1,1,&rank_left,&rank_right);
```

which shifts on the x axis (second input) of 1 cell (third input). Similarly, to find the top and bottom neighbour we use:

```
MPI_Cart_shift(comm_cart,0,1,&rank_top,&rank_bottom);
```

which, in this case, shifts on the y axis - i.e., vertically - of 1 cell.
Now that we know the neighbours of each processor, we have to exchange the correct data between them. We know that, since C stores data in row-major order, we have no problem in exchanging rows of a matrix because the data are contiguous in memory. The problem occurs when we want to exchange data that are not contiguous in memory. Fortunately, MPI gives us the routine `MPI_Type_vector` which creates a vector-type object. In order to use it in the correct manner, we first have to understand how it works. As input it accepts:

- *count*: number of blocks that we want to send;

- *blocksize*: dimension of the block (in our case, since we want to send only one column of doubles, it is equal to 1);
- *stride*: distance in memory between elements that we want to send;
- *type*: type of the item that we want to send.

The last element in `MPI_Type_vector` is the output. Now we have created only a structure without any data inside it. When we use it in `MPI_Send`, what it does is to take *count* elements with distance from each other equal to *stride*.
The last step is the exchange of the correct data between neighbours. In order to do so, I used the MPI routine `MPI_Send` and `MPI_Recv` in which, for each processor, I specify the correct position of the data that has to be sent and the position in which the receiver has to put the data received. As explained before, I used the vector-type object only to exchange data between left and right neighbours. The code I implemented is the following:

```
// to the top
MPI_Send(&data[DOMAINSIZE+1],SUBDOMAIN, MPI_DOUBLE, rank_top, 0,
    MPI_COMM_WORLD);
MPI_Irecv(&data[DOMAINSIZE*(DOMAINSIZE-1)+1],SUBDOMAIN, MPI_DOUBLE,
     rank_bottom, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);

// to the bottom
MPI_Send(&data[DOMAINSIZE*(DOMAINSIZE-2)+1], SUBDOMAIN, MPI_DOUBLE,
     rank_bottom,0,MPI_COMM_WORLD);
MPI_Irecv(&data[1], SUBDOMAIN, MPI_DOUBLE, rank_top, 0,
    MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);

// to the left
MPI_Send(&data[DOMAINSIZE+1], 1, data_ghost, rank_left, 0,
    MPI_COMM_WORLD);
MPI_Irecv(&data[DOMAINSIZE*2-1], 1, data_ghost, rank_right, 0,
    MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);

// to the right
MPI_Send(&data[DOMAINSIZE*2-2], 1, data_ghost, rank_right, 0,
    MPI_COMM_WORLD);
MPI_Irecv(&data[DOMAINSIZE], 1, data_ghost, rank_left, 0,
    MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);
```

3. Parallelizing the Mandelbrot set using MPI [20 Points]

With the aim of creating the Mandelbrot set using MPI, I firstly implemented the functions *createPartition()*, *updatePartition()* and *createDomain()* in the file *consts.h* as follows:

```
Partition createPartition(int mpi_rank, int mpi_size)
{
    Partition p;
```

```
    // TODO: determine size of the grid of MPI processes (p.nx, p.ny),
       see MPI_Dims_create()

    int gridSize[2] = {0,0}; // automatically select how many process
       in x and y
    MPI_Dims_create(mpi_size, 2, gridSize); // distribution of process
       in x and y
    p.nx = gridSize[1];
    p.ny = gridSize[0];
    // p.ny = 1;
    // p.nx = 1;


    // TODO: Create cartesian communicator (p.comm), we do not allow
       the reordering of ranks here, see MPI_Cart_create()
    int periods[2] = {1,1};
    MPI_Cart_create(MPI_COMM_WORLD, 2, gridSize, periods, 0, &p.comm);
    // MPI_Comm comm_cart = MPI_COMM_WORLD;
    // p.comm = comm_cart;


    // TODO: Determine the coordinates in the Cartesian grid (p.x, p.y)
       , see MPI_Cart_coords()
    int coords[2];
    MPI_Cart_coords(p.comm, mpi_rank, 2, coords);
    p.y = coords[0];
    p.x = coords[1];

    //p.y = 0;
    //p.x = 0;

    return p;
}

/**
Updates Partition structure to represent the process mpi_rank.
Copy the grid information (p.nx, p.ny and p.comm) and update
the coordinates to represent position in the grid of the given
process (mpi_rank)
*/
Partition updatePartition(Partition p_old, int mpi_rank)
{
    Partition p;

    // copy grid dimension and the communicator
    p.ny = p_old.ny;
    p.nx = p_old.nx;
    p.comm = p_old.comm;

    // TODO: update the coordinates in the cartesian grid (p.x, p.y)
       for given mpi_rank, see MPI_Cart_coords()
    int coords[2];
```

```
    MPI_Cart_coords(p.comm, mpi_rank, 2, coords);
    p.y = coords[0];
    p.x = coords[1];

    //p.y = 0;
    //p.x = 0;

    return p;
}

/**
Structure Domain represents the information about the local domain of
    the current MPI process.
It holds information such as the size of the local domain (number of
    pixels in each dimension - d.nx, d.ny)
and its global indices (index of the first and the last pixel in the
    full image of the Mandelbrot set
that will be computed by the current process d.startx, d.endx and d.
    starty, d.endy).
*/
Domain createDomain(Partition p)
{
    Domain d;

    // TODO: compute size of the local domain
    d.nx = IMAGE_WIDTH/(p.nx);
    d.ny = IMAGE_HEIGHT/(p.ny);

    //d.nx = IMAGE_WIDTH;
    //d.ny = IMAGE_HEIGHT;

    // TODO: compute index of the first pixel in the local domain
    d.startx = (p.x * (IMAGE_WIDTH)/p.nx);
    d.starty = (p.y * (IMAGE_HEIGHT)/p.ny);

    //d.startx = 0;
    //d.starty = 0;

    // TODO: compute index of the last pixel in the local domain
    d.endx = (p.x + 1)*(IMAGE_WIDTH/p.nx)-1;
    d.endy = (p.y + 1)*(IMAGE_HEIGHT/p.ny)-1;

    //d.endx = IMAGE_WIDTH - 1;
    //d.endy = IMAGE_HEIGHT - 1;

    return d;
}
```

As we can see in Fig. 1, the time for the computation of the Mandelbrot set using two processors is halved compared to the time using only one processor. In contrast, the difference between the time using 4 processors and the time using 8 processors is almost the same. We can notice that the computation time of, as example, processor 5, in the run using 16 processes, is exactly equal to the computation time of processor 9. This comparison can be extended to all the processes. The reason for this behaviour lies in the symmetry, with respect to the x axis, of the Mandlebrot

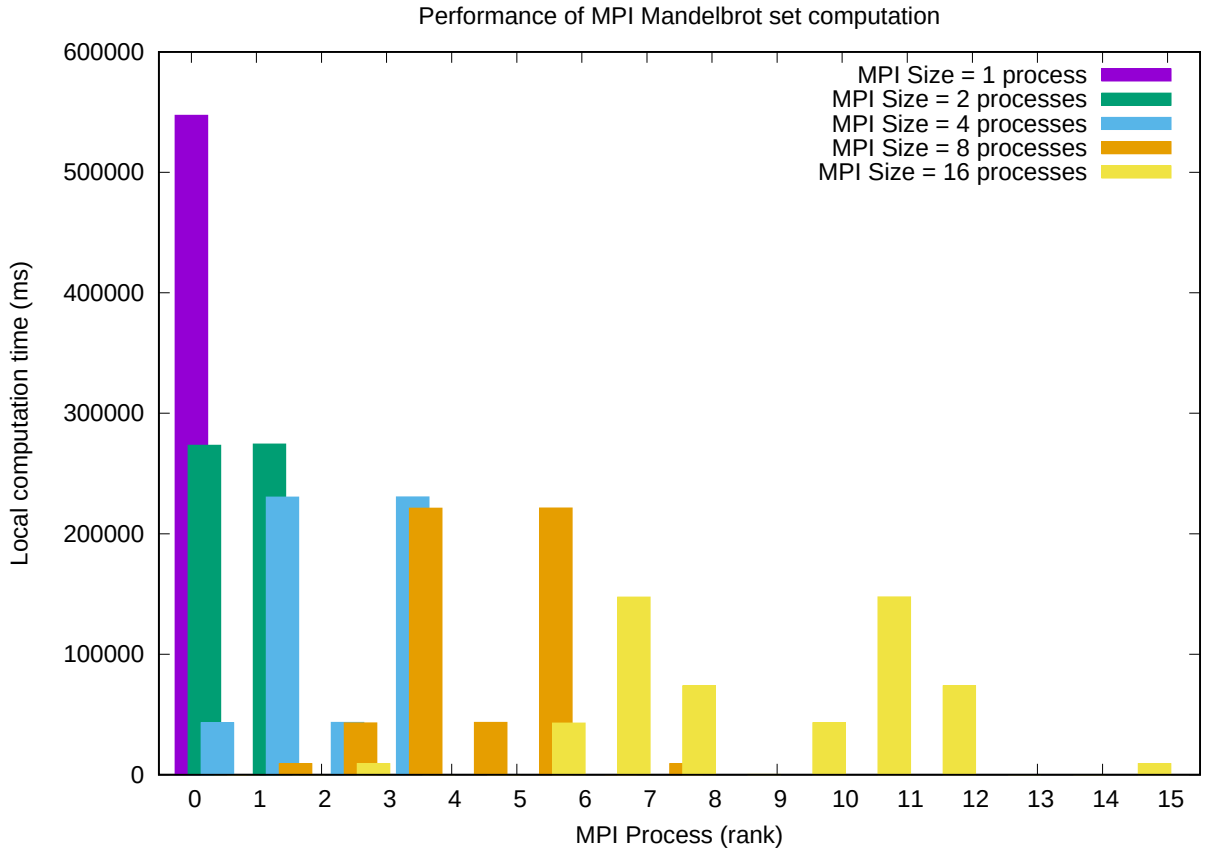

Figure 1: Performance with different amount of processors

set. Thus, in the 4×4 grid reported in the left side of Fig.2, the pairs of processors with the same computation time are: (0 - 12), (1 - 13), (2 - 14), (3 - 15), (4 - 8),(5 - 9),(6 - 10) and, (7 - 11). We know that different areas of the Mandelbrot set require a different amount of time to be computed. In my opinion, the most interesting aspect - which is also the explanation of why this method of partitioning is not the most efficient - is the difference in terms of computation time between different processes. One of the most efficient ways to parallelize the Mandelbrot set consists in distributing the work among the processes in an equal way in terms of computation time.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

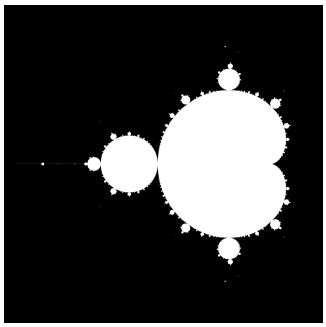

Figure 2: Left: grid of 16 processors. Right: Mandelbrot set

4. Option A: Parallel matrix-vector multiplication and the power method [50 Points]

With the aim of parallelizing the power method using MPI, I principally wrote two files. The first one, called *consts.h*, contains the functions that we need in order to do the computations - e.g. matrix-vector multiplication or the norm of the vector - and the second one is the *main.c* file, which uses the function in *consts.h* and gives the timing for the computation of the power method. I decided to divide the work in the following way:

1. Create a function called *generateMatrix* which generates - initially - a test matrix with fixed entries for debugging purposes and then a matrix with random numbers;
2. Create a function called *generateVector* which generates - initially - a test vector with fixed entries for debugging purposes and then a vector with random numbers;
3. Create a function called *norm* which divides each entry of a given vector for its norm;

After the creation of the above functions, I focused my effort on the parallelization of them using MPI. My first issue was the distribution of the rows of the matrix between the processors. In order to do so, instead of generating a complete matrix, I generated only the part of the matrix associated with each processor: thus, e.g., with a 8×8 matrix with $p = 2$, the processor 0 uses the function *generateMatrix* to generate its own 16 entries (and the same happens for the second processor). The function is the following:

```
double *generateMatrix(int n, int size)
    {
    double *A;
    double randNum;
    int i;

    A = (double*)calloc(n * (n/size), sizeof(double));
    srand(time(NULL));

    for (i=0; i< n*(n/size); i++){
        randNum = rand();
        A[i] = randNum;
    }

    return A;
}
```

The second function that I implemented is *generateVector* which, as the name suggests, generates a vector of length equal to the number of rows/colmuns of the matrix.

```
double *generateVector(int n)
{
    double *x;
    int i;
    double randNum;
    x = (double*)calloc(n, sizeof(double));

    srand(time(NULL));

    for (i = 0; i < n; i++){
        randNum = rand();
        x[i] = randNum;
```

```
    }
    return x;
}
```

The most important function is the one that performs the matrix-vector multiplication using the MPI routines `MPI_Bcast` and `MPI_Gather` to exchange information between the processors. I implemented it in the following way:

```
double *matVec(double *x, double *A, int size, int n, int my_rank)
{
    int i,j, ierr;
    double sum, res[n/size];

    ierr = MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    if (ierr != MPI_SUCCESS){
        printf("Error in MPI_Bcast\n");
        MPI_Abort(MPI_COMM_WORLD, 3);
    }

    for (i = 0; i<(n/size); i++){
        sum = 0;
        for (j = (i*n); j<n*(i+1); j++){
            sum += A[j] * x[j-(i*n)];
        }
        res[i] = sum;
    }

    if(my_rank==0){
        ierr = MPI_Gather(&res, (n/size), MPI_DOUBLE, x, (n/size),
            MPI_DOUBLE, 0, MPI_COMM_WORLD);

        if (ierr != MPI_SUCCESS){
            printf("Error in MPI_Gather\n");
            MPI_Abort(MPI_COMM_WORLD, 4);
        }
    }else{
        MPI_Gather(&res, (n/size), MPI_DOUBLE, NULL, 0, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
    }

    return x;
}
```

The main function that puts all the pieces together is the following:

```
#include "consts.h"
#include "hpc-power.h"
#include "hpc-power.c"

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

```
#include <math.h>


int main(int argc, char* argv[])
{
    // Initialize MPI
    MPI_Init(&argc, &argv);
    int my_rank, size, n, k;

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Check if the number of proessors used is correct
    if (my_rank == 0){
        int flag = 1;
        if (size == 1 || size == 4 || size == 8 || size == 12 || size ==
            16 || size == 32 || size == 64){
            flag = 0;
        }
        if(flag){
            printf("Error: number of processors must be equal to
                {1,4,8,12,16,32,64}! \n");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
    }

    // Check if the number of processor is a perfect divisor of n
    n = atoi(argv[1]);
    if (n % size != 0){
        printf("Error: the number of processors must be a perfect divisor
             of n! \n");
        MPI_Abort(MPI_COMM_WORLD, 2);
    }

    // Initialization of the random matrix A
    double *A = generateMatrix(n, size);

    // Initialization of the random vector x
    double *x = generateVector(n);

    k = atoi(argv[2]);

    // start timer
    double initTime = hpc_timer();

    for (int i = 0; i< k; i++){
        // Do the power method k times
        x = powerMethod(x, A, size, n, my_rank);
    }

    // Stop the timer
    if(my_rank==0){
        double finalTime = hpc_timer();
```

```
        double time = finalTime - initTime;
        printf("%f\n",time);
    }

    free(A);
    free(x);
    MPI_Finalize();
    return 0;
}
```

4.1. Results

In this section I will present the result of the parallel matrix-vector multiplication and of the power method, focusing my attention on both a strong and weak scaling analysis.

- **Strong scaling analysis**: As we can see in the left part of Fig.3, an increase in the number of processors used - initially - is extremely beneficial but, due to the increasing number of communication, the efficiency of using more processors decreases drastically from 16 to 32 processes and is more than halved from 32 to 64 (left part of Fig.3).

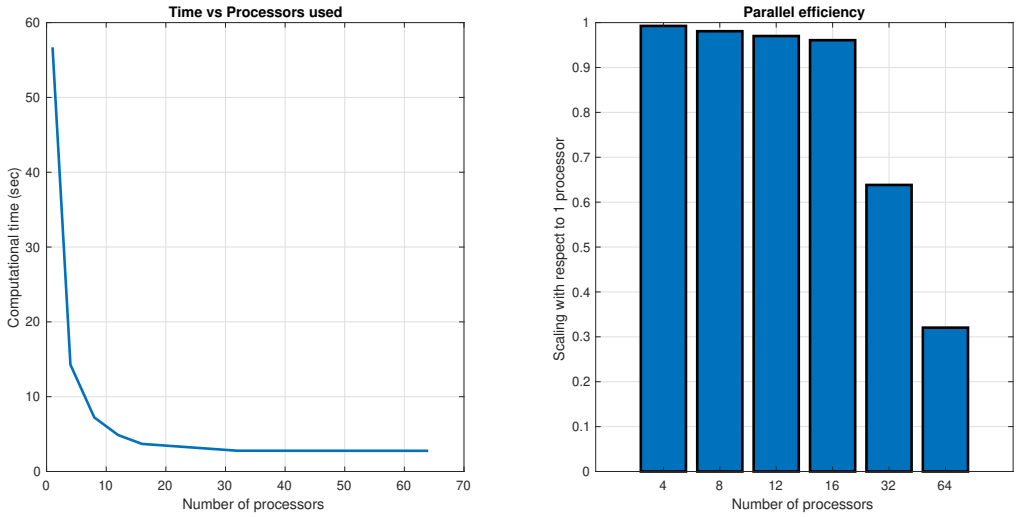

Figure 3: Left: comparison between computation time and number of processors used. Right: parallel efficiency.

- **Weak scaling analysis**: In order to perform a weak scaling analysis, I run my code using different number of processors and matrix sizes which are proportional to $\sqrt{p}$. In Table 1 I reported, for each number of processors used, the matrix size. I fixed the number of iterations of the power method to 100. In order to find the correct matrix size, I used the formula:

$$n = 4000\sqrt{p} - \mathrm{mod}(4000\sqrt{p}, p).$$

In Fig.4, I reported the results of the weak scaling analysis. As we can clearly see, when we increase the number of processors used from 32 to 64, the computational time is rocketing, due to the time needed for the communication between processors. In order to achieve the maximum of the performance, we have to take into consideration also the communication time which - for large matrices - has an enormous impact on the computation time.

# of proccessors	# of rows/columns
1	4000
4	8000
8	11312
12	13848
16	16000
32	22624
64	32000

Table 1: Number of processor use associated with the correct matrix size.

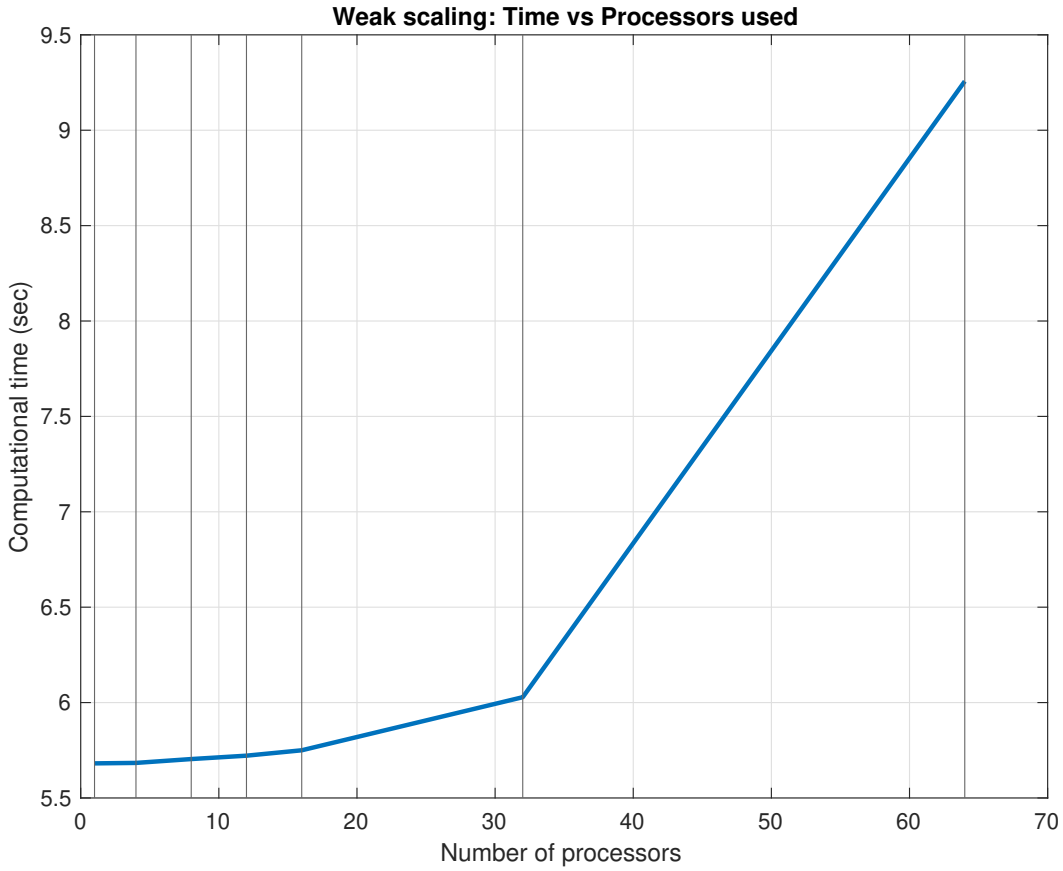

Figure 4: Weak scaling of matrix vector multiplication and power method

4.2. How to run the main file

In order to run the program on your laptop with, e.g., 4 processors, n = 4000, and k = 100, you have to follow the instructions reported below:

```
make
mpirun -np 4 ./main 4000 100
```