Università della Svizzera italiana | Institute of Computing CI

High-Performance Computing Lab **Institute of Computing**

Student: Fabian Gobet Discussed with: Ekkehard Steinmacher, Henrique Gil, Francesco Gnan

Solution for Project 4

1. Ring maximum using MPI [10 Points]

The operation to cycle between elements in a cyclical manner, side-by-side, can be represented in terms of the modulo operation over the number of processes. Hence, for a given process i we can get the left neighbour and right neighbour through the following code

```
right = (my_rank+1)%(size);
left  = (my_rank-1)%(size);
```

I then initialize the *snd_buf* for each element and also assign that value to variable max

```
snd_buf = (3*my_rank)%(2*size);
max = snd_buf;
```

Because we want all processes to retain the maximal value amongst all, and since they communicate in a cyclic manor (sending to the right, receiving from the left) we have to do as many iterations as the number of processes. To achieve this I use MPI_Send() and MPI_Recv() statements in the following manner

```
for(int i = 0; i<size; i++){
    MPI_Send(&snd_buf,1,MPI_INT,right,0,MPI_COMM_WORLD);
    MPI_Recv(&rcv_buf,1,MPI_INT,left,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    if(rcv_buf>max)
        max=rcv_buf;
    snd_buf = rcv_buf;
}
```

By the end of the cycle, each process should retain in the *max* variable the maximum value amongst all.

2. Ghost cells exchange between neighboring processes [15 Points]

In order to create 2-dim cartesian communicator I will make use of the function MPI_Cart_create(), hence some variables must be defined before. I will also make use of the function MPI_Dims_Create() to automatically set up the dimensions according to the number of processes as the dimension (2D).

The MPI_Dims_Create() takes in as arguments the number of processes, the dimension size and an int array of the same size of the dimensions. For values of the array set to 0, the function will reassign them in order to fit all processes in a fitting manner. Hence, the code to achieve this is the following

```
dims[0] = 0;
dims[1] = 0;
MPI_Dims_create(size, 2, dims);
```

Next, to use the function MPI_Cart_create() we also need to define the periods for each dimension, that is an int array which will specify if each dimension is cyclical or not (1 or 0, respectively), and a *reorder* int variable which indicates if the mapping between processes and their ranking should be redefined. We don't want the ranks to be reassigned, hence *reorder* will be 0. Furthermore, we want both dimensions to behave in a cyclical manner, hence both components of the *periods* variable will be 1.

Upon creating the cartesian map, we need integrate each process in the new comm world with respect to the cartesian mapping. The code to achieve all this is the following

```
periods[0] = 1;
periods[1] = 1;
int reorder = 0;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &comm_cart);
MPI_Comm_rank(comm_cart, &rank);
```

To assign the neighbouring ranks for each rank, I made use of the function MPI_Cart_shift(). This function takes as input the comm world, the dimension for which we want to explore the neighbours, the displacement number between neighbours and the variables where we want to store the results. We need to execute this code 2 times, one for each dimension and for both of the respective adjacent ranks. Hence the code to achieve this is the following

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

To create a derived datatype with respect to a data structure, namely a matrix in row major vector form, I made use of the function MPI_Type_vector(). This function takes as input the number of elements to be sent in a block, the number of blocks to be sent, the spacing between each element in the data structure (stride), the data type for each elements and the variable reference where we want to store this derived datatype. Sending contiguous values of a row is easily achieved without the need of this data structure, but sending contiguous values in a column needs to make use of the said data type (given the row major vector form).

Because our data structure is of size $DOMAINSIZE$x$DOMAINSIZE$, the spacing between elements in the same column will be $DOMAINSIZE$. Furthermore, the number of elements we want to send is $SUBDOMAIN$. Once this derived data type is created, we must commit it so we can effectively use it in communications. Hence, the code for this is as follows

```
MPI_Type_vector(SUBDOMAIN, 1, DOMAINSIZE, MPI_DOUBLE, &data_ghost);
MPI_Type_commit(&data_ghost);
```

We are now in conditions of implementing the message exchange between neighbouring processes given the cartesian map. As such, the following characteristics should be taken into account:

1. Each process sends data in a direction and proceeds to receive data from the opposite direction (i.e. send up, receive from bottom)
2. Sending up or down equates to sending a row, and then receiving a row.
3. Sending up implies that the process receiving the data will store the data in the bottom part of its array. The revse holds.
4. The values to be sent up start at index $DOMAINSIZE + 1$ and are stored by the receiving process starting at index $DOMAINSIZE * (DOMAINSIZE - 1) + 1$.
5. For bottom send the start index is $DOMAINSIZE * (DOMAINSIZE - 2) + 1$ and the starting receiving index is 1.

6. The same idea applies to sending left and right, but now we use the derived datatype to allocate the sending and the receiving data.

7. For left send the start index is $DOMAINSIZE + 1$ and the receiving starting index is $2 * DOMAINSIZE - 1$

8. For right send the start index is $2 * DOMAINSIZE - 2$ and the receiving starting index is $DOMAINSIZE$

Given the prior statements, the code to achieve these communications looks as follows

```
//  to the top
MPI_Send(&data[DOMAINSIZE+1],SUBDOMAIN,MPI_DOUBLE,rank_top,0,comm_cart);
MPI_Recv(&data[DOMAINSIZE*(DOMAINSIZE-1)+1],SUBDOMAIN,MPI_DOUBLE,rank_bottom,0,
    comm_cart,&status);
//  to the bottom
MPI_Send(&data[DOMAINSIZE*(DOMAINSIZE-2)+1],SUBDOMAIN,MPI_DOUBLE,rank_bottom,0,
    comm_cart);
MPI_Recv(&data[1],SUBDOMAIN,MPI_DOUBLE,rank_top,0,comm_cart,&status);
//  to the left
MPI_Send(&data[DOMAINSIZE+1],1,data_ghost,rank_left,0,comm_cart);
MPI_Recv(&data[2*DOMAINSIZE-1],1,data_ghost,rank_right,0,comm_cart,&status);
//  to the right
MPI_Send(&data[2*DOMAINSIZE-2],1,data_ghost,rank_right,0,comm_cart);
MPI_Recv(&data[DOMAINSIZE],1,data_ghost,rank_left,0,comm_cart,&status);
```

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

3.1. createPartition()

Like in the previous exercise, to create a cartesian grid I first define an int array for the dims initialized to 0 set its values using the MPI_Dims_create() function, then reassigning the values of $p.nx$ and $p.ny$ respectively

```
int dims[2] = {0};
MPI_Dims_create(mpi_size, 2, dims);
p.ny = dims[0];
p.nx = dims[1];
```

Then again I make use MPI_Cart_create() with periods set to 0 and reorder set to 0

```
int periods[2] = {0};
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &(p.comm));
```

Finally, to get the corresponding coordinates i use the function MPI_Cart_coords() that takes as input the communication world, the rank process, the number of dimensions and the variable address to store the calculated values. It is worth noting that the size of the array to store the values must be equal to the number of dimensions. Hence the following code achieves this

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

3.2. updatePartition()

The partition update consists of only reassignments of values and and a new computation over coordinates using MPI_Cart_coords() function (previously explained)

```
// copy grid dimension and the communicator
p.ny = p_old.ny;
p.nx = p_old.nx;
p.comm = p_old.comm;
// update the coordinates in the cartesian grid (p.x, p.y) for given mpi_rank
```

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

3.3. createDomain()

The dimensions for which each process will through the x and y axis can be given by the division between the image width and the number of elements in the 0 dimension (i.e. x), and the division by the image height by the number of elements in the 1 dimension (i.e. y)

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

Following the previous idea, we then know that *startx* and *starty* can be calculated through the use of the coordinates and the *d.nx* and *d.ny* values

```
d.startx = (p.y)*(d.nx);
d.starty = (p.x)*(d.ny);
```

Finally, the *endZ* can simply be given by the sum of the respective *startZ* with *d.nZ* minus 1, for Z = {x,y}. Hence the following codes achieves this

```
d.endx = (d.startx)+(d.nx)-1;
d.endy = (d.starty)+(d.ny)-1;
```

The number of elements to be sent, and received, by each element can be given by $(d.nx)*(d.ny)$. Hence, when rank is not 0, the process will execute the following code

```
MPI_Ssend(c,(d.nx)*(d.ny),MPI_INT,0,0,p.comm);
```

And when the rank is 0 it will execute

```
MPI_Status status;
MPI_Recv(c,(d1.nx)*(d1.ny),MPI_INT,proc,0,p.comm,&status);
```

In the prior segment I used MPI_Ssend(), a blocking operation, to ensure that the sending process doesn't terminate before the receiving process has received its data. This ensures that upon termination and de-allocation of memory blocks, the data has already been processed.

Upon running *run_perf.sh*, I got the following graph from the resulting *perf.ps* file

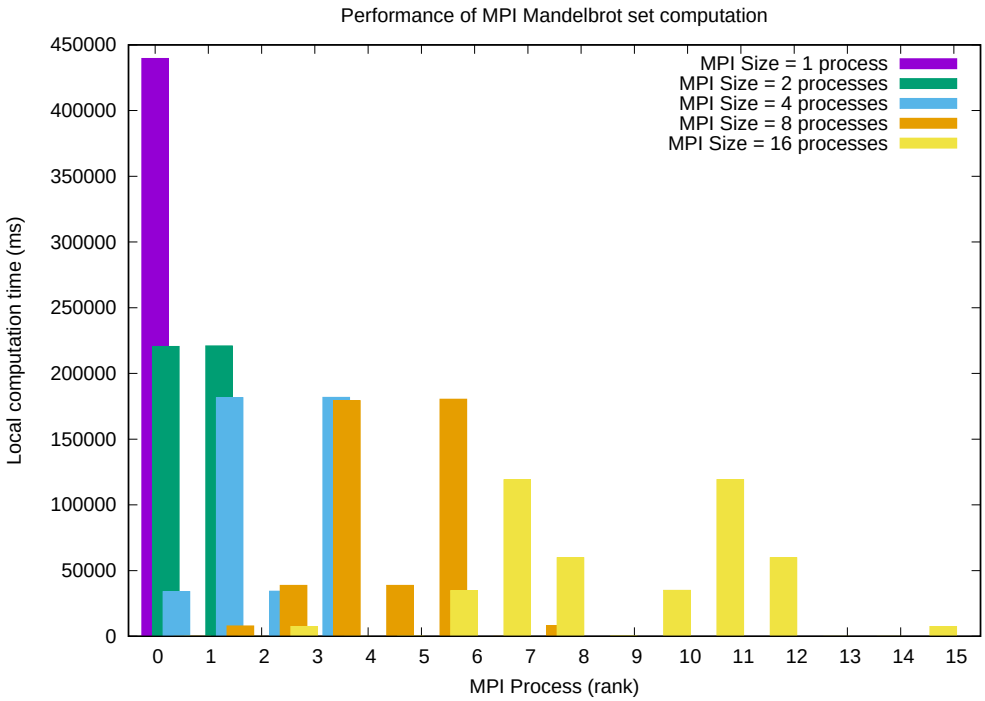

Figure 1: Mandel exercise, performance graph

By analyzing the previous graph we can immediately notice that running the code on only one process takes more than double of the time regarding any other number of processes. Consequentially, we can also observe that for increasing number of processes the computational time tends to decrease. This happens because each process is calculating it's own part of the mandel set in a parallel fashion, thus equating to a boost in computational performance per unit of time and a lowering of the computational time.

We can also observe that for each given number of processes above 2, the load balance for each process is not equal. This happens because some processes are assigned to a block of the mandel set in which there's a lower factor of numbers which belong to the set (corresponding mostly to edges of the image, the black spots).

4. Option A: Parallel matrix-vector multiplication and the power method [40 Points]

4.1. hpc_generateMatrix()

For this exercise I first started by defining the hpc_generateMatrix() function in the const.c file. We want this function to return an allocated block of memory of the size of $n*numrows*sizeof(double)$, initialized to 'n' in each component (could also be 1). It is important no notice that this implementation was built upon the idea that each process will only have a slice of the matrix, whereas the given code had an input argument *startrow* which suggested otherwise.

```
double* hpc_generateMatrix(int n, int numrows) {
    double* A = (double*)calloc(n * numrows, sizeof(double));
    for (int i = 0; i < n*numrows; i++)
        A[i] = 1;
    return A;
}
```

4.2. norm()

Next, I defined a function to calculate the norm given an address for the vector of doubles and its size 'n', in the powermethod.c file.

```
double norm(double *vector, int n){
    double sum = 0.0;
    for(int i=0; i<n; i++)
        sum += vector[i]*vector[i];
    return sqrt(sum);
}
```

4.3. matvec()

The matvec() function takes in as arguments the address to the matrix slice (in row major form), the number of columns and rows, an address to the vector by which to perform mat mult, and lastly an address to an array of size of the number of rows to store the values of the result.

```
void matvec(double* A, int n, int numrows, double* vector, double* result){
    for(int i=0; i<numrows; i++){
        double innerproduct = 0.0;
        for(int j=0; j<n; j++)
            innerproduct += A[i*n+j]*vector[j];
        result[i]=innerproduct;
    }
}
```

4.4. randVector()

In order to initialize an allocated vector of a given size randomly, for a given range, I implemented the following function which then returns the address for the resulting vector

```
double* randVector(int size, double minRange, double maxRange){
    srand((unsigned)time(NULL));
    double* vector = (double *)(malloc(size*sizeof(double)));
        for(int i = 0; i<size; i++)
            vector[i] = minRange + ((double)rand() / RAND_MAX) * (maxRange -
                minRange);
    return vector;
}
```

4.5. powerMethod()

This function in the core of this implementation. The powerMethod() function takes in as arguments the addresses for the matrix slice, the vector for mat mult and the vector to store the results, the rank of the process, the number of rows and columns and a MPI communication channel.

The architecture of this implementation consists in the normalization of the vector and spread of information by the process ranked 0. The reason for this centralization is because the opposed possibility, where all processed calculate the norm of the vector, would imply many more communications between all processes and redundant calculations.

Hence, the function starts by calculating the norm if the rank of the process is 0 and then broadcasting it for the rest of the processes. On the other hand, the other processes await for this information to arrive.

Once each process has the vector, they proceed to do execute matvec() and then send the result to process of rank 0. The process of rank 0 in turn awaits for this information and concatenates it automatically in the previous vector address using the properties of the function MPI_Gather(), which is specifically designed for collective communication and centralized data gathering.

```
void powerMethod(double* A, double* vector, double* result ,int my_rank, int n, int
    numrows, MPI_Comm comm){
```

```
    if(my_rank==0){
        double nrm = norm(vector,n);
        for(int i=0; i<n; i++)
            vector[i] = vector[i]/nrm;
    }
    MPI_Bcast(vector, n, MPI_DOUBLE, 0, comm);
    matvec(A, n, numrows, vector, result);
    if(my_rank!=0)
        MPI_Gather(result, numrows, MPI_DOUBLE, NULL, 0, MPI_DOUBLE, 0, comm);
    else
        MPI_Gather(result, numrows, MPI_DOUBLE, vector, numrows, MPI_DOUBLE, 0, comm
            );
}
```

4.6. main()

In the main body I start by initializing MPI and assigning the values to *size* and *my_rank* in the usual manner, and asserting that the number of columns and rows (possibly given as input) is divisible by the number of processes to ensure equal distribution.

```
int my_rank, size;
int n = 9600;
double start, end;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if(argc>1)
    n = atoi(argv[1]);
if(n%size!=0){
    printf("please run this with 'n' divisible by number of processors\n");
    MPI_Finalize();
    exit(1);
}
```

Once this assertion is done, i calculate the number of rows per process, and initialize the matrix slices, the vector variable and the result variable. Furthermore, if the rank is 0, then the vector is initialized to a random vector of numbers (double) of size of the number of columns, and a timer is initialized.

```
int numrows = n/size;
double* A = hpc_generateMatrix(n, numrows);
double* vector = (double *)(malloc(n*sizeof(double)));
double* result = (double *)(malloc(numrows*sizeof(double)));
if(my_rank==0){
    vector = randVector(n, -5, 5);
    start = hpc_timer();
}
```

We are now in conditions of starting the iterations of the powerMethod() function. Once All of the iterations are complete, the process of rank 0 proceeds to stop the timer and run hpc_verify() to check whether the result is congruent with the expected value. At the very end, all processed free up the memory allocations.

```
int numrows = n/size;
for(int i=0; i<100; i++){
    powerMethod(A, vector, result , my_rank, n, numrows, MPI_COMM_WORLD);
}
if(my_rank==0) {
    end = hpc_timer() - start;
    int correct = hpc_verify(vector, n, end);
    //printf("Result %s, in %f seconds.\n", correct==0 ? "incorrect":"correct",end);
    printf("%d,%d,%f\n", n, size, end);
}
free(A);
free(result);
free(vector);
```

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

4.7. Strong scaling

In order to perform a strong scaling analysis, I set the number of columns (and rows) to 9600 and ran the the powermethod program for 1, 4, 8, 12, 16 and 32 numbers of processes.

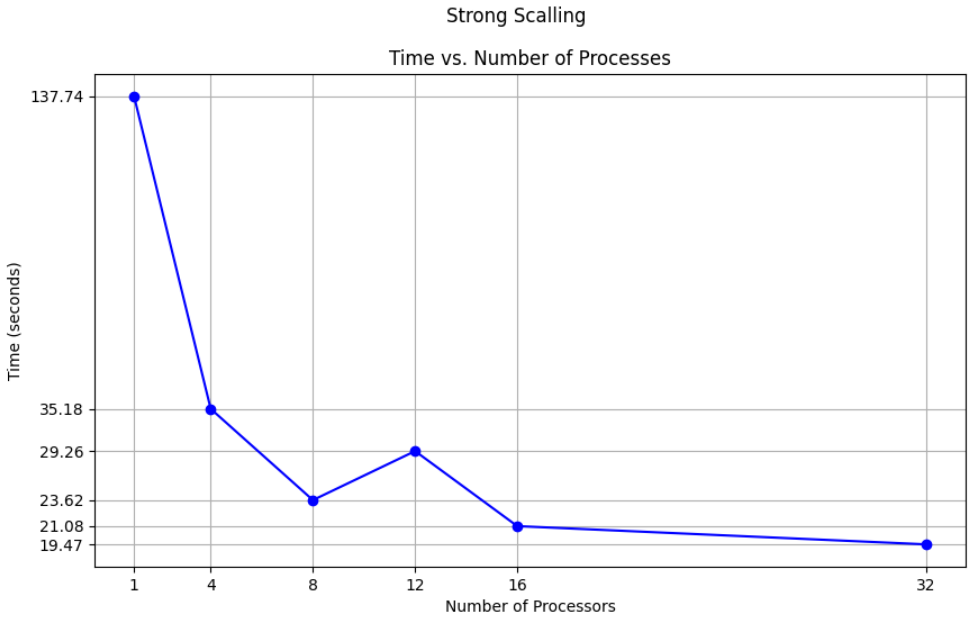

Figure 2: Powermethod exercise, strong scaling

By analysing the prior graph, in which the y-scale is logarithmic, we can observe that there is exponential improvement in times with an odd case at 12 processes. This goes to show that indeed parallelization brings a lot of benefit when considering the overall computational time.

But if we were to consider the efficiency as the factor relation between the time for one process only divided by the multiplication of the number of processes by the overall respective time, we may see that efficiency per process tends to go down.

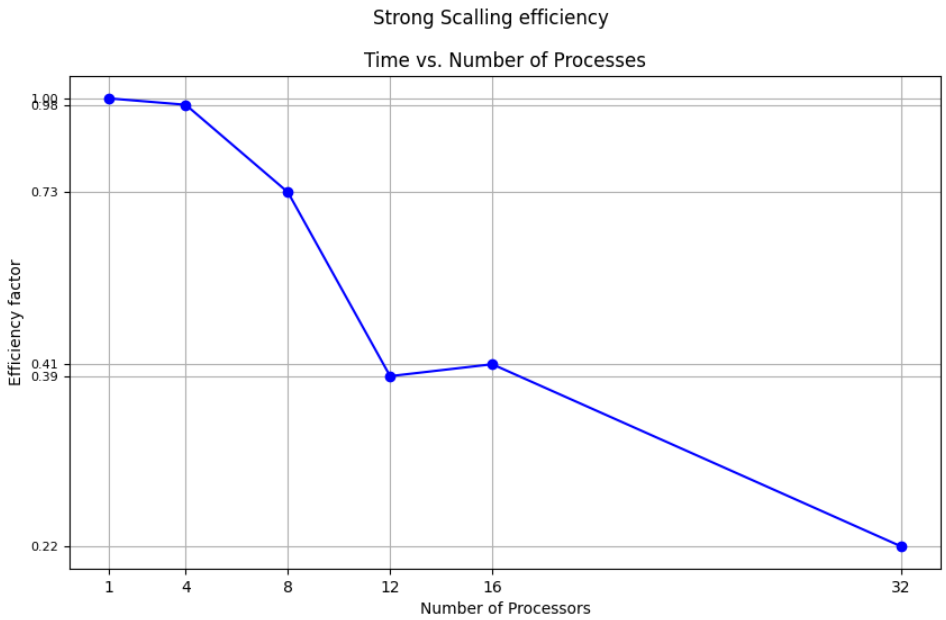

Figure 3: Powermethod exercise, strong scaling efficiency

This of course is congruent with the fact that the sum of times per process, assuming equal workload, is in total bigger than the time for only one process. Furthermore, the more processes we use, the lower the effieciency tends to be.

4.8. Weak scaling

When performing a weak scalling analysis we can assess how the implementation performs on a constant workload per processor and determine where there are hardware bounds for which the implementation starts to increasingly perform worse.

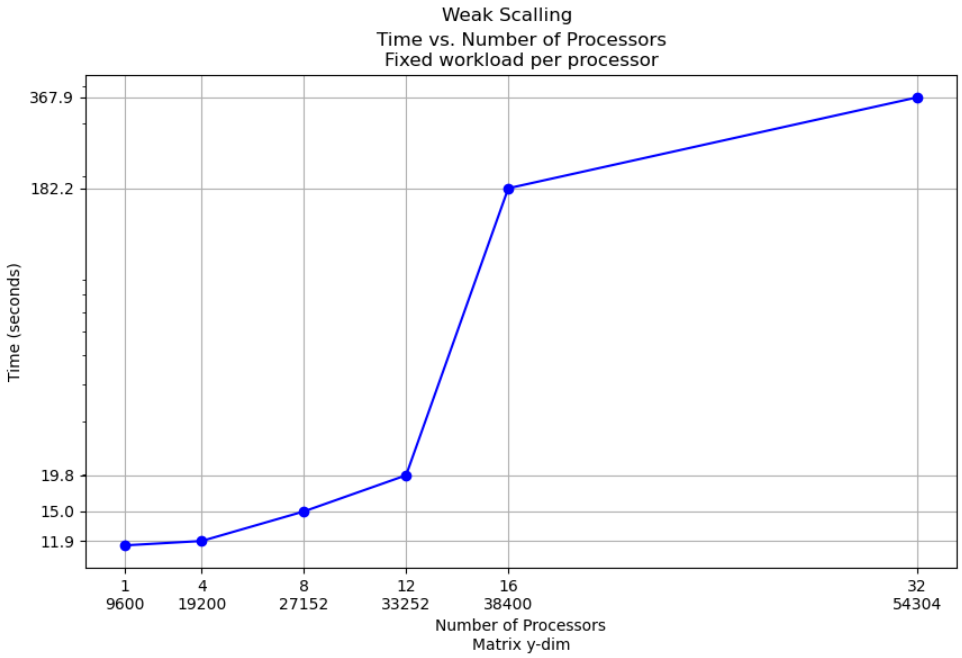

Figure 4: Powermethod exercise, weak scaling

Because this implementtation was executed on a 12 core device (MacBook 16 Pro), we can see that for more than 12 processes there is a great increase in time which is not so proportional to prior increases. This implies that for more than 12 processes there is a lot of overhead regarding the management of threads in a 12-core device.

In similar fashion to strong scaling analysis, efficiency drops with the scaling of number of processes, more so because the work load remaining constant for each process.

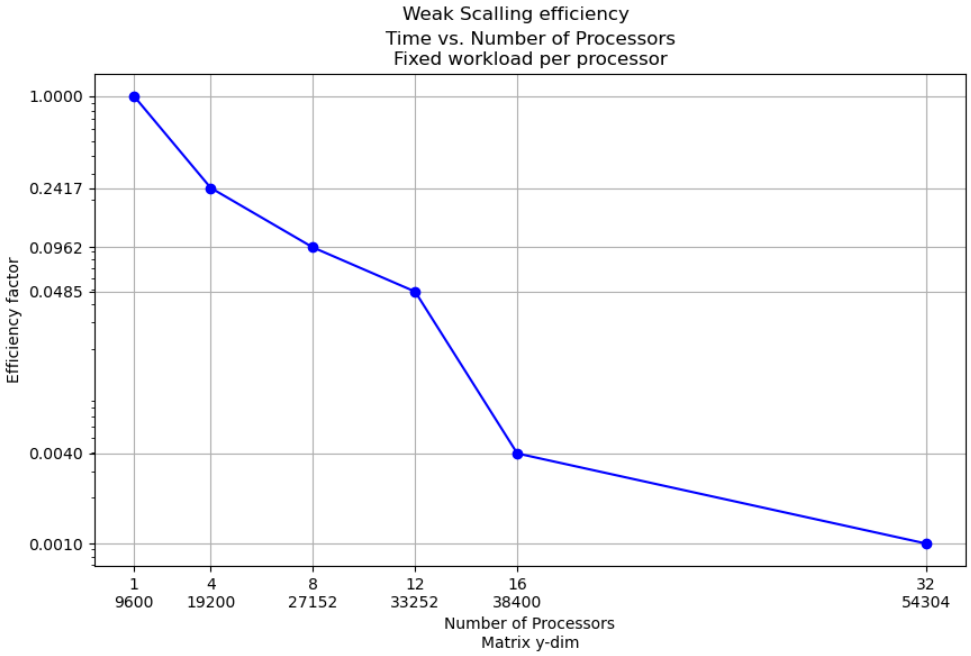

Figure 5: Powermethod exercise, weak scaling efficiency