Lawrence Asamoah Adu-Gyamfi (1484610), Maximilian Grunwald (1519529)

MPI - Performance Analysis

Contents

1 Introduction

In this assignment we again work on the solution of a 2D Laplace equation using Jacobi Iteration method, this time with means of parallelization with MPI (Message Passing Interface). Utilizing the functions provided by MPI it well be possible to achieve a parallelized solution where the different process can communicate with each other, sharing data to improve efficiency and speed of our program. For this, we will use a remote access to execute the calculation on a cluster, which allows us to use up to 12 processes on two nodes.

2 Communication between processes using MPI

2.1 The basic version

Starting from the serial solution to the laplace problem provided, we used `MPI` commands to achieve a communication between the processes. We attach the full commented code and focus on the most important added commands here.

2.1.1 Laplace step and laplace initialization

```
#include <mpi.h>
...

float laplace_step(float *in, float *out, int n, int me, int nprocs, int
   ri, int rf)
{
  int i, j;
  float error=0.0f;

  if(me==0){
    ri+=n;
  }
  else if (me==nprocs -1){
    rf-=n;
  }

  for ( j=ri; j < rf; j+=n ){
  for ( i=1; i < n-1; i++ ){
    out[j+i]= stencil(in[j+i+1], in[j+i-1], in[(j-n)+i], in[(j+n)+i]);
    error = max_error( error, out[j+i], in[j+i] );
  }
  }

  return error;
}
```

In comparison to earlier versions, our laplace step function in the `MPI` case gains complexity. We have to take onto account the subdivision of our matrix as well as the additional rows at the start and at the end.
For this, we have to send not only the addresses of the input and output matrix and the problem dimension, but also the number of processes, the current process id and the positions where the first and the last actual rows of a submatrix start.
Depending on wheter we are in the first or the last process, we shift the first/last row (`ri`/`rf`)

by one row. This helps us to avoid overwriting of the boundary conditions. Than, we perform the double for-loop going from `ri` to `rf` and updating all but the first and the last element of a row with the stencil function. As in earlier versions, the error gets updated in every iteration and returned at the end.

```c
void laplace_init(float *in, int n, int rows_pp, int me)
{
  int i;
  const float pi  = 2.0f * asinf(1.0f);
  memset(in, 0, n*(rows_pp+2)*sizeof(float));
  for (i=0; i<rows_pp; i++) {
    float V = in[(i+1)*n] = sinf(pi*(me*rows_pp+i) / (n-1));
    in[(i+1)*n+( n -1) ] = V*expf(-pi);
  }
}
```

Although we work with several processes, the initialization following the given boundary conditions has to be done globally. So we need to send the information about the current process id to the function `laplace_init`.
Having this information, each process fills his own part of the boundary conditions as distinct from former versions where we could fill the matrix with its initial values globally and only once.
Another option to handle this would have been to let only the first process bother about the initalization. That would have allowed us to take the same code as in earlier versions. But on the other hand there would have been the need to communicate this initial values to the other process.

2.1.2 int main

```c
int main(int argc, char** argv){
...
// we introduce a MPI_Status variable and initialize the MPI communication
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  //Initial Row and Final Rows indices
  //beginning of the first actual row: skipping the k additional rows
  int ri = n;
  //beginning of the last actual row
  int rf = n * n/nprocs;

  //Rows per process
  int rows_pp = n / nprocs ;

  //reserve space for the actual matrices plus space for k additional rows
  A    = (float*) malloc( n*(rows_pp + 2)*sizeof(float) );
  temp = (float*) malloc( n*(rows_pp + 2)*sizeof(float) );

int iter = 0;
  while ( g_error > tol*tol && iter < iter_max )
  {
    iter++;

  //for all but the first process:
  if (me > 0){
    MPI_Send(A+ri,n, MPI_FLOAT,me-1,0, MPI_COMM_WORLD);

    MPI_Recv(A, n, MPI_FLOAT, me-1, 1, MPI_COMM_WORLD, &status);
  }

  // for all but the last process:
  if (me < nprocs -1){

    MPI_Send(A+rf, n, MPI_FLOAT, me+1, 1, MPI_COMM_WORLD);

    MPI_Recv(A+rf+n, n, MPI_FLOAT, me+1, 0, MPI_COMM_WORLD, &status);
  }

  l_error= laplace_step (A, temp, n, me, nprocs, ri, rf);

  // swap pointers A & temp
    float *swap= A; A=temp; temp= swap;
    MPI_Allreduce(&l_error, &g_error, 1, MPI_FLOAT, MPI_MAX,
        MPI_COMM_WORLD);
  }
  //conclude the MPI communication
  MPI_Finalize();
  }
```

In the `int main`, we have to arrange the `MPI` communication. First we initialize the `MPI` communication, set the positions of each process' first and last row and allocate enough memory for the problem dimension plus two additional rows. Having this, our goal is to interchange the information of each process with its precedent and subsequent process.

The communication is done by the functions `MPI_Send` and `MPI_Recv`. Both functions take as arguments the starting address from where data should be copied, the number of copied elements, the `MPI` data type (here `MPI_FLOAT`), the id of the destinating process, a tag (0 or 1), the `MPI` communicator and the address of the status variable.

We have to ask in an `if` condition for the exeption cases, when we are in the first process (no precessor to communicate with) or in the last process (no successor to communicate with). After exchanging the information, we call the `laplace_step`, produce an error and synchronize the error between all of the processes using `MPI_Allreduce`.
After the maximal number of iterations is reached or the error fell below the tolerance, we conclude the `MPI` communication via `MPI_Finalize`.

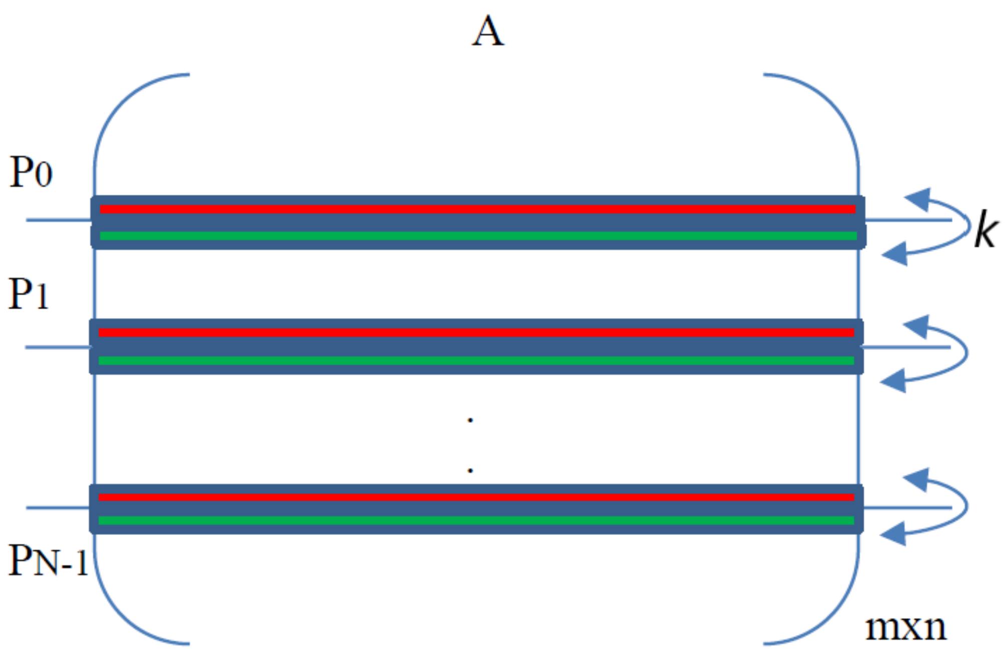
2.2 Improvements to reduce communication overhead

After having implemented the basic `MPI` version, the next point to look at is the reduction of communication overhead. While we can linearly reduce and share the calculation work that has to be done in the laplace step by increasing the number of processes on more cores, this will have a effect on the communication.

Having a rising number of processes means to have a higher amount of communication between them. But this is a problem we can tackle by thinking about methods the reduce the communication done by each process. Different possibilities were proposed:

- **Block partitioning** For this reduction method the total number of communicated elements decreases, while the number calls of `MPI_Send`/`MPI_Recv` increases. For example, having 4 processes, it will be send/received $2n$ elements instead of $3n$ for the basic version.
  On the other hand, `MPI_Send`/`MPI_Recv` is called 4 times instead of 3 times. We first started to work on the block partitioning. But we became aware that keeping track of all needed special cases when it comes to sending and receiving of information between the processes is very complicated and could easily be a source of errors. Also the initialization and the laplace step itself had to be handled with a lot of care, which could lead to potential errors.

- **Block communication** We decided to work on this optimization method since this promises to reduce the number of needed `MPI_Send`/`MPI_Recv` by k. We noticed that using `MPI_Send`/`MPI_Recv` was not longer successful for a higher problem size (from $n > 500$). So we decided to use non-blocking communication which allowed us to work on higher problem dimensions.



- **Hybrid solution** We did not approach this solution, although it seemed to be highly interesting. But given the complexity of the implementation and the obstacles we encountered when implementing the more basic versions we opted not to go for this solution.

2.3 On the reduction method via Block Communication

We want to explain our reasoning when implementing the block communication optimization. Again, we attached the complete code and focus on the main differences in comparison to the basic MPI version.
At cost of calling the `MPI_Send`/`MPI_Recv` less often, the amount of calculation in the `laplace_step` function rises. That is, we do the laplace step like in the basic MPI version, but iterate this step k (number of communicated rows) times.

2.3.1 Laplace initialization

```
void laplace_init(float *in, int n, int rows_pp, int me, int k)
{
  int i;
  const float pi  = 2.0f * asinf(1.0f);
  memset(in, 0, n*(rows_pp+2*k)*sizeof(float));
  for (i=0; i<rows_pp; i++) {
    float V = in[(i+k)*n] = sinf(pi*(me*rows_pp+i) / (n-1));
    in[(i+k)*n+( n -1) ] = V*expf(-pi);
  }
}
```

The only difference here in comparison to the basic version is the shift by k rows when we write the boundary conditions into the matrix.

2.3.2 int main

```
int main(int argc, char** argv)
{
...
int k = 2;
...
//beginning of the first actual row: skipping the k additional rows
  int ri = n * k;
//beginning of the last actual row
  int rf = n * n/nprocs + (n * (k-1));
 ...
 A    = (float*) malloc( n*(rows_pp + 2*k)*sizeof(float) );
  temp = (float*) malloc( n*(rows_pp + 2*k)*sizeof(float) );
...
  //set boundary conditions
  laplace_init (A, n, rows_pp, me, k);
  laplace_init (temp, n, rows_pp, me, k);
  if(me==0){
  A[((n/128)+k)*n +n/128] = 1.0f; // set singular point
  }
...
MPI_Request request;
int iter = 0;
  while ( g_error > tol*tol && iter < iter_max )
  {
    iter++;
    if (((iter - 1) % k) == 0){
    if (me > 0){
      // send first rows
      MPI_Isend(A+ri,k*n, MPI_FLOAT,me-1,0, MPI_COMM_WORLD, &request);
      // Receive last rows( last but one from me - 1)
      MPI_Irecv(A, k*n, MPI_FLOAT, me-1, 1, MPI_COMM_WORLD, &request);
    }
    if (me < nprocs -1){
      MPI_Isend(A+rf-(k-1)*n, n*k, MPI_FLOAT, me+1, 1, MPI_COMM_WORLD, &
         request);
      MPI_Irecv(A+rf+n, n*k, MPI_FLOAT, me+1, 0, MPI_COMM_WORLD, &request)
         ;
    }
    MPI_Wait(&request, &status );
    //perform the laplace step k times with different start and end rows
    for(int cnt = 0; cnt < k; cnt++){
      l_error= laplace_step (A, temp, n, me, nprocs, ri-((k-1)-cnt)*n, rf
         +((k-1)-cnt+1)*n);
    }
      float *swap= A; A=temp; temp= swap;
    MPI_Allreduce(&l_error, &g_error, 1, MPI_FLOAT, MPI_MAX,
       MPI_COMM_WORLD);
    }
  }
MPI_Finalize();
```

We followed the idea of sharing k rows each time we call `MPI_Send`/`MPI_Recv`. For higher values of n we encountered a missbehaviour of the execution (possibly a deadlock). A good solution for this problem was to use the nun-blocking versions of the former communication functions. So with `MPI_Isend`/`MPI_Irecv`, we got rid of the problems.

When now the k additional rows of a eachs process' submatrix are filled, it needs k iterations of the `laplace_step` to update the values with all the given information.
In each iteration, we decrease our calculation region by one row at the top and at the bottom, so after k steps, we used all the k communicated rows and updated all entries in the inner matrix.
This is necessary to provide the values in the inner matrix with updated values even in the kth iteration. Only by going more far away from the actual matrix and solve the stencil also on $k-1$ additional rows allows us to obtain the right values in the end.

3 Results

In this section we present and interpret the results we obtained when we executed our programs with different problem sizes and varied number of processes. Most results were produced by the version witch uses block communication. However, the results in this section concern a version of our code, where we didn't performed the laplace step k times after each sending, but used the basic procedure to update the values of the matrices.

3.1 The Basic version

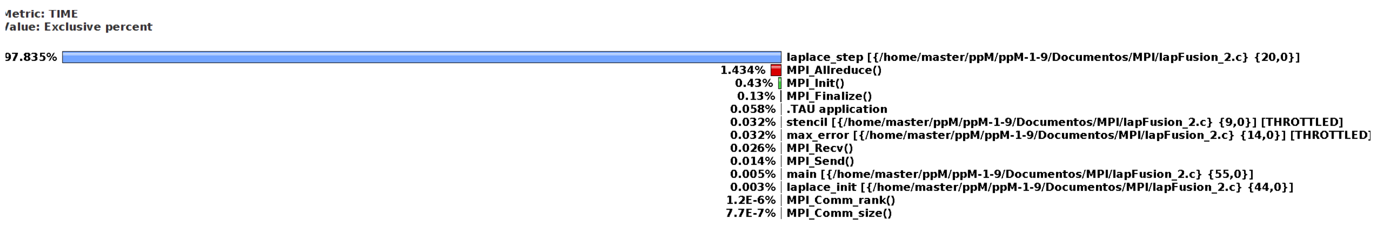


Figure 1: paraprof output for the basic version: splitted up for all program parts

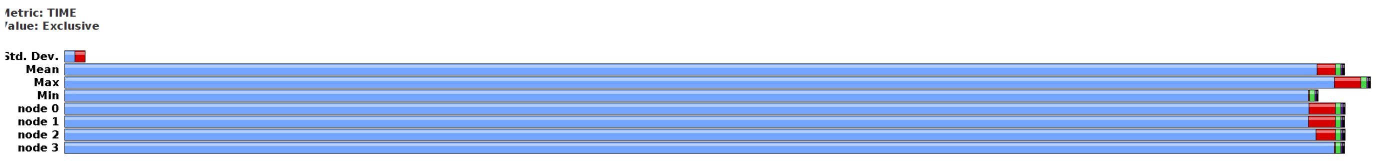


Figure 2: paraprof output: exclusive comparison of the execution with 4 nodes

We proceed to present some results of the optimized version (using block communication) as mentioned above.

3.2 Changing numbers of processes

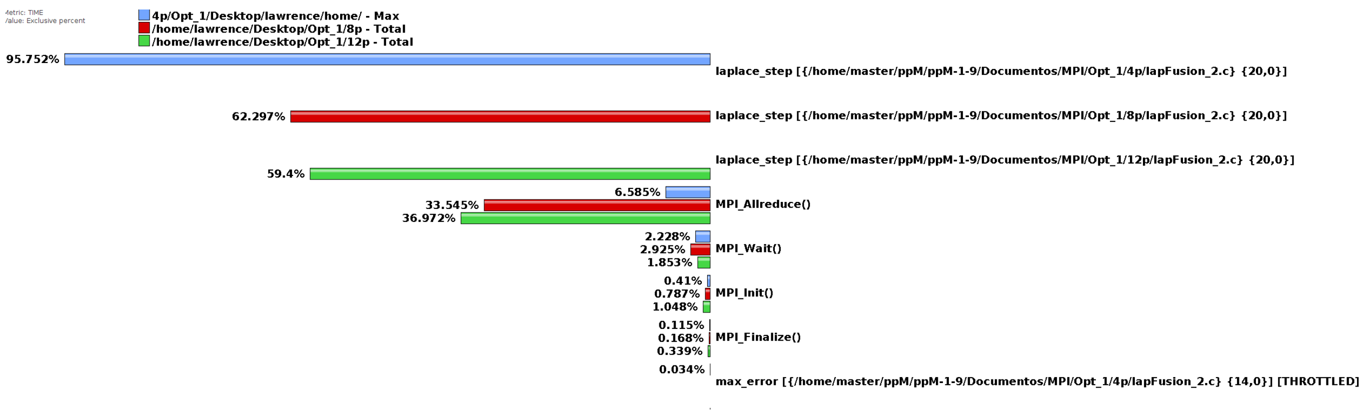


Figure 3: paraprof output: behaviour under different number of processes

In the figure 12 we compare the runtime for 4 (blue), 8 (red) and 12 (green) processes. While there can be seen a clear reduction on the execution time of the `laplace_ step`, we have to register a rise of the duration of the `MPI_Allreduce`. All other parts of the program which are not shown in the graphic required to little run time, that we neglected them in our observation. This holds especially for the functions `MPI_Send` and `MPI_Recv`. Probably the dimension of our problem is still to small to produce a remarkable overhead when it comes to communication. It can be seen that the times needed for `MPI_Init` as well as the `MPI_Finalize` rise with a higher number of processes.

3.2.1 Imbalance

With respect to load balance we noticed this worked well for 4 processes as shown by figure 4. However for the case of 8 and 12 processes, there seemed to be imbalance of work especially

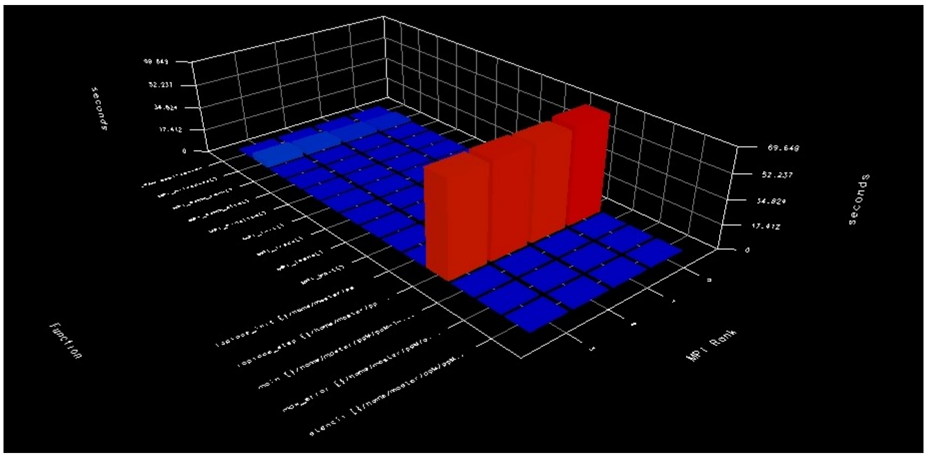


Figure 4: paraprof output for 4 processes

for the `laplace_step` and `MPI_Allreduce` functions. This is shown in the figures below.

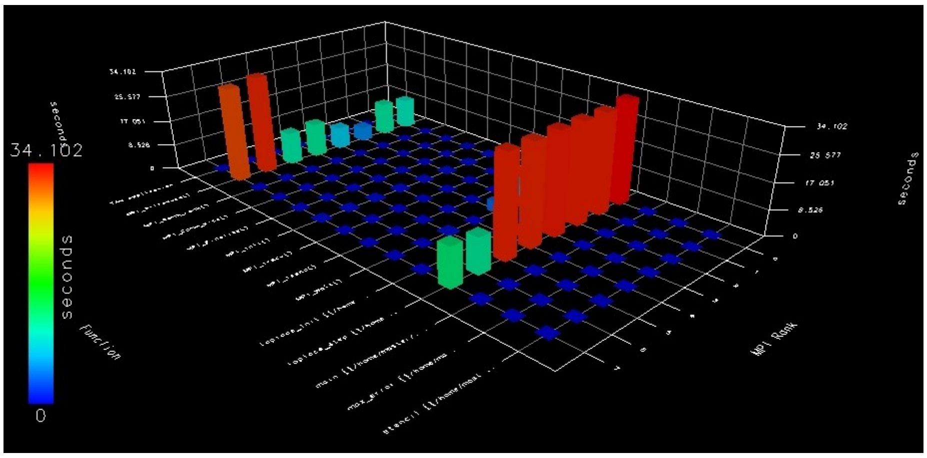


Figure 5: paraprof output for 8 processes on 2 nodes

An interesting observation was that the time spent for these functions seem to compensate each other, meaning if a process finished earlier with the laplace step, they proceeded to work on the `MPI_Allreduce` computations and communications.
A can be seen in figure 7, the last 4 processes finished the laplace calculation (blue) earlier and were concerned with the `MPI_Allreduce` (wine red) from this moment on. The pink colour for the 9th process denotes `MPI_Wait`.

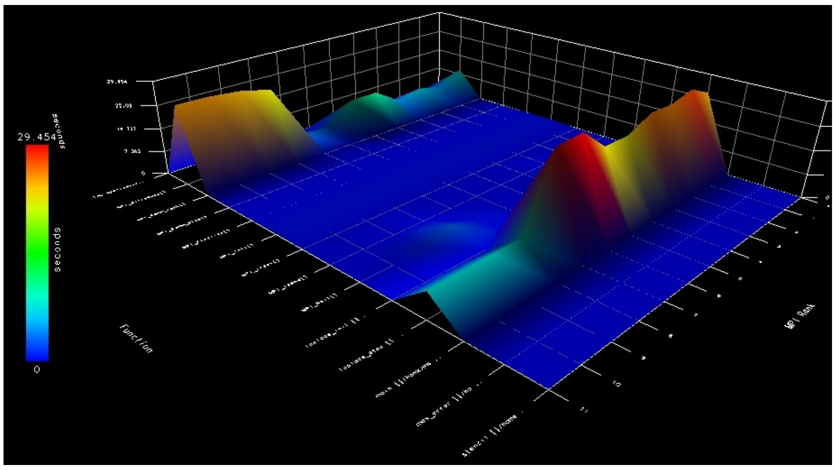


Figure 6: paraprof output for 12 processes on 2 nodes

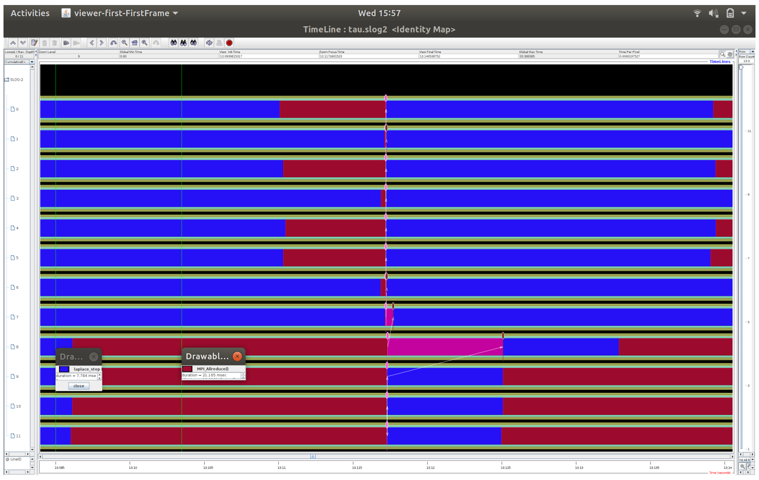


Figure 7: slog output for 12 processes

Before continuing with the next `laplace_step`, this process waits for the communicated information (depicted by the arrow) of its neighbour process.

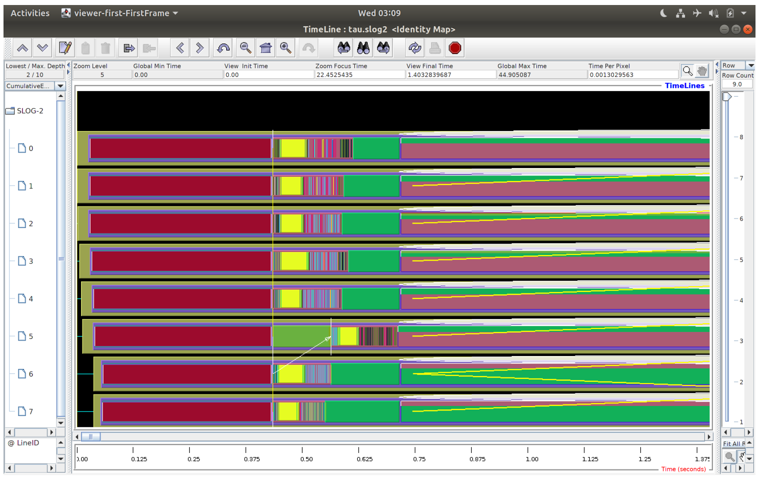


Figure 8: jumpshot: 8 processes

3.2.2 The communication matrix

It was a task to have a look at the communication matrix, which can be produced by `TAU` ans visualized by `paraprof`. The pictures below serve us as an confirmation of how we imagined the communication between the processes.

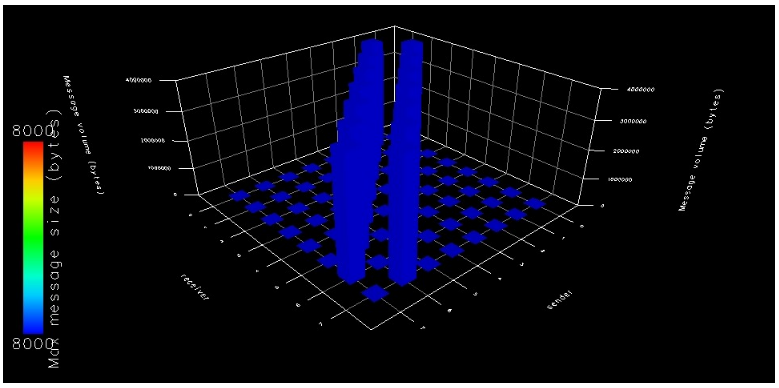


Figure 9: The 3D communication matrix for 8 processes

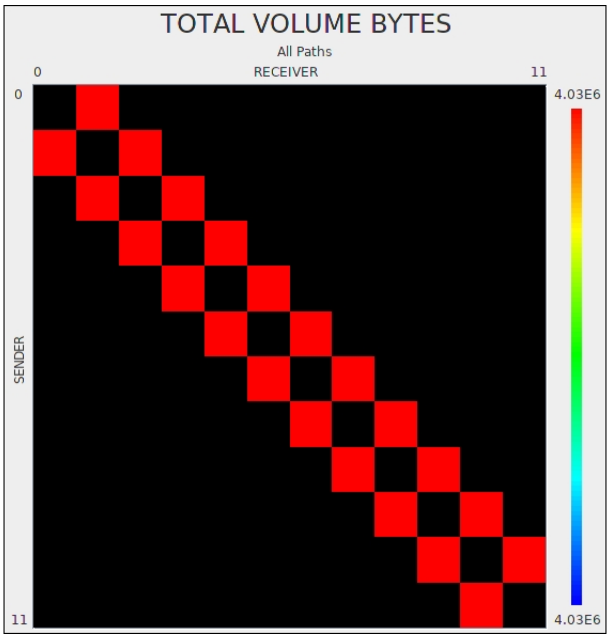


Figure 10: The plane communication matrix for 12 processes

In figure 10 we see the visualized communication matrix, mapping the amount of communication betwen `Sender` and `Receiver`. As we expect and implemented, all processes communicated a constant amount of data (all squares are red) to their left and right neighbours. Exceptions are the first process (which can only communicate with its successor) and the last process (which only has an predecessor to communicate with).

3.3 Changing the problem size n

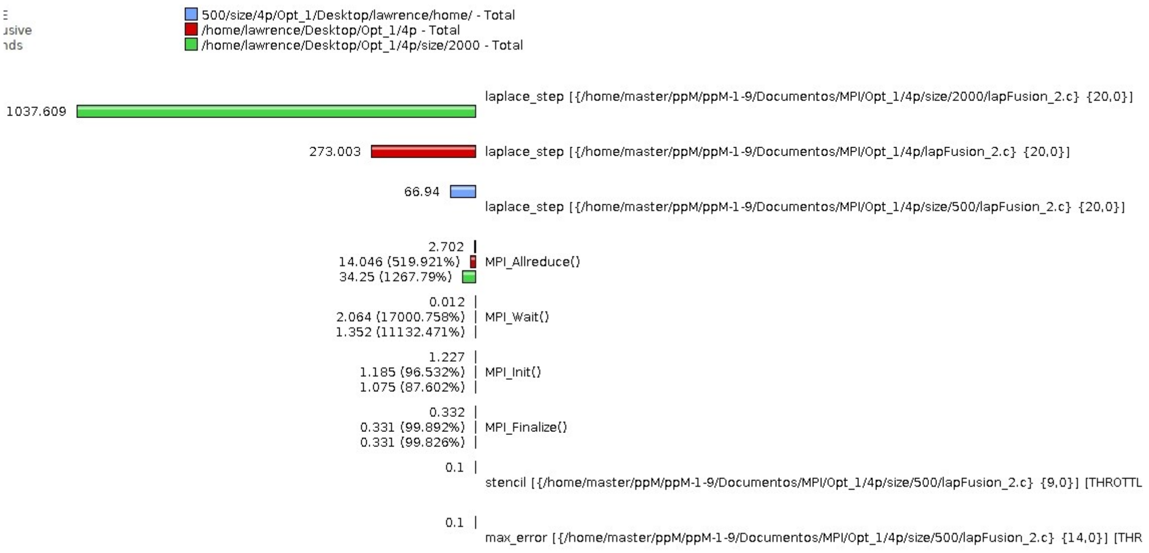


Figure 11: paraprof output: behaviour under different number of processes

When we changed the problem size, we noticed a quadratical decrease of run time. We executed the program with 4 processes and for $n = 500$, $n = 1000$, $n = 2000$. Whenever we halved the problem size, the execution time shrank by the factor 4. This is mainly due to this exactely behaviour of the `laplace_step`. The second largest factor on execution time, `MPI_Allreduce`, showed a more irregular decrease.

3.4 Changing the communication size k

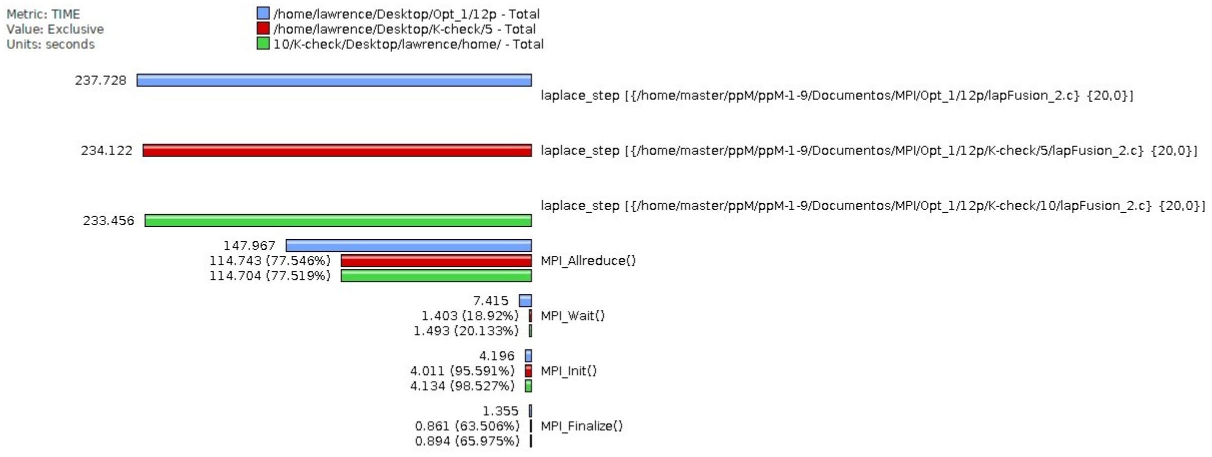


Figure 12: paraprof output: behaviour for different values of k: $k = 2, 5, 10$

The variation of the communication size k did not have a major effect, presumedly this would have a more remarkable effect for bigger problem dimensions.

4 Conclusion

As part of this exercise, we have implemented parallelism in the basic version of the Laplace equation using Jacobi Iteration method through Message Passing (`MPI`). We have further implemented an optimisation of the base `MPI` version which initially distributed one row to an adjoining process. The optimisation sought to send more than 1 row during each communication step and does reduce the amount of time spent in communicating the rows. We have analysed the performance of the various implementations of the code using performance analysis tools such as paraprof and jumpshot from `TAU`.

As part of the optimisation we have had to implement non-blocking communication as we noticed a possible situation of deadlock using point-to-point as utilised in the base code. This modification allowed to run the code for larger problem sizes and for a higher number of processes. We noticed instance of load imbalance especially as the number of processes increased. This phenomenon was more prominent for the functions which used up most of the execution time.

Interestingly, these cases seemed to compensate each other meaning, probably the processes jumped to the next function if they finished the previous earlier. We noticed as well that the percentage of the execution time used for the initialisation of the matrix increased with the increase in processors which is logical. This scenario was similar for the `MPI_Init` and `MPI_Finalise` functions as well.

An increase in problem size keeping the same number of processes resulted in a corresponding quadratic increase in the execution time for the expensive functions. (`laplace_step` and `MPI_Allreduce`). In terms of the optimisation implemented, this did not have a major impact on reducing the execution time and a possible explanation could be the fact that a major percentage of the execution time was not spent on the communication and therefore a decrease in this parameter did not have a huge impact on the overall execution time. Also increasing the amount of rows communicated during each each communication cycle did not have a major impact on reducing the execution time and this can also be attributed to the latter explanation.