Parallel Programming

MPI Parallelization Assignment

Miquel Barcelona (NIU: 1359817)

Pauline Martens (NIU: 1561215)

17 December 2019

Introduction

In this assignment, we will parallelize and optimize the solution to the 2D Laplace equation on a cluster of nodes by using MPI. MPI or Message Passing Interface is a communication protocol for parallel programming, specifically used to allow applications to run in parallel acrross several separate computers that are connected by a network. The following command is used to compile each version of the parallelized Laplace problem assuming that we are working with the version 3.0.0 of the openmpi compiler.

```
$ mpicc -o lap lapFusion.c -lmpi -lm
```

In the case of the hybrid version we must include the library `omp.h` so we use the following command:

```
$ mpicc -o lap lapFusion.c -fopenmp -lmpi -lm
```

More information about the compilation process can be found in the `Makefile` file included. Anyway, the following line is used to execute the code if we want to set 4 processes.

```
$ mpirun -np 4 lap (size) (iter_max)
```

All the versions that we will present in the current project have been checked using the base line version of the code for a given size and a given number of iteration. In particular, we use the output error of a problem of size 1000 and 100 maximum iterations.

Code explanation

In the following section we explain the different strategies used to parallelize the base version of the Laplace equation using MPI as the communication and synchronization mechanism. In addition, we also present two different optimizations of the MPI base version implemented looking for improving the performance and reducing the effects of the communication costs.

MPI Implementation

First of all, MPI is implemented in the base code to parallelize the problem. In this way, the main strategy is based on dividing the matrix A of size $n \times n$ in N different blocks, where we define N as the number of processes and we assume that n can be divided by N. Then, each one of the processes, `nprocs`, will take care of compute in each iteration step his own blobk P_i. For that reason, we have to initialize the MPI communications.

```
  /* Initialize MPI */
  int me, nprocs;
  MPI_Init(&argc, &argv);

  double t1 = MPI_Wtime();
  MPI_Comm_rank( MPI_COMM_WORLD, &me);
  MPI_Comm_size( MPI_COMM_WORLD, &nprocs);
```

Due to the data dependency of the Laplace equation, each block P_i needs the the bottom and upper rows of its neighbour blocks P_{i-1} and P_{i+1} respectively. Because of this, each of the N blocks will have size $n/N + 2$ in order to save this two additionals rows.

```
  /* Allocating memory */
  A    = (float*) malloc( (n/nprocs + 2) * n * sizeof(float) );
  temp = (float*) malloc( (n/nprocs + 2) * n * sizeof(float) );
```

In this step we need to change also the `laplace_init()` function as we have changed the size of the old matrix A by the new partition blocks P_j. Then, the main loop of this function will be:

```
  for (i=0; i < n/nprocs; i++) {
#define R(j) ((j)*n/nprocs)  // j means actual process number
    float V = in[(i+1)*n + 0/*col*/] = sinf(pi*(i + R(me)) / (n-1));
    in[ (i+1)*n+n-1 ] = V*expf(-pi);
  }
```

Note that we have defined the macro $R(j)$, being j the number of the process, in order to be able to make the index transformation between the row of the block P_j and the row of the total matrix A.

Then, we rewrite the main loop of the Laplace equation. In this loop we need to perform the data interchanging between the processes in each iteration before being able to execute the `laplace_step()` function. In this communications we have to consider the index of the block P_j because the first and last block of the matrix A do not need to compute the first and last row respectively. Overall, we need to perform a reduction among all the processes to obtain the maximum error of each one of them.

```
while ( gerror > tol*tol && iter < iter_max )
{
  if (me < nprocs - 1) {
      MPI_Send(A + RF, n, MPI_FLOAT, me + 1, 33, MPI_COMM_WORLD);
  }
  if (me > 0) {
      MPI_Recv(A, n, MPI_FLOAT, me - 1, 33, MPI_COMM_WORLD, &status);
  }
  if (me > 0) {
      MPI_Send(A + R0, n, MPI_FLOAT, me - 1, 33, MPI_COMM_WORLD);
  }
  if (me < nprocs - 1) {
      MPI_Recv(A + RF + n, n, MPI_FLOAT, me + 1, 33, MPI_COMM_WORLD, &status);
  }
  iter++;
  error = laplace_step (A, temp, n /*col*/, me, nprocs);
  MPI_Allreduce(&error, &gerror, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
  float *swap = A; A = temp; temp = swap; // swap pointers A & temp
}
```

In this case, the macros `R0` and `RF` define the starting position of the first and last row of the partition blocks that need to be sent and they allow us to use pointer arithmetic. In this sense, `A + R0` will point to the second row of the block and `A + RF` points to the penultimate row.

Finally, we modify the `laplace_step()` function in order to take care of the new matrix indexes. In this case, the calculations will take part only from the second row of the block P_j to the penultimate row which is the row number $n/N + 1$. This is performed only in the inner partitions. In the case of the first and last blocks we do not need to compute the borders of the matrix A because they remain fixed.

```
float laplace_step(float *in, float *out, int n/*col*/,
                   int me, int nprocs)
{
  int i, j;
  float error=0.0f;
  int i_first = 1, i_last = n/nprocs + 1;
  if (me == 0) i_first++;
  if (me == nprocs - 1) i_last--;
#define N n/nprocs + 2 /* Number of rows of the new matrix */
  for ( j = i_first; j < i_last; j++ ) /* rows */
    for ( i = 1; i < n - 1; i++ ) /* cols */
    {
      out[j*n+i]= stencil(in[j*n+i+1], in[j*n+i-1], in[(j-1)*n+i], in[(j+1)*n+i]);
      error = max_error( error, out[j*n+i], in[j*n+i] );
    }
  return error;
}
```

We will refer to this version thorugh this project as *Version 1*.

Optimization of parallelized problem

After parallelizing the laplace step equations using MPI and the above explained strategy, we will look at possible optimizations of the obtained version. We are looking for reducing the overhead introduced by the communications between processes. In this section we present two different optimizations.

Overlapping communication and computation This optimization entails that first the inner points of the matrix portion assigned to each process will be computed, and after that the border points. This offers the advantage that a process can start computing the inner points at any given time and does initially not depend on the data received from other processes because there is no data dependency. As such, communication overhead is minimized by overlapping the communication [1].

In this way, we only need to change the interchanging data part of the last version of the code and defining two new functions called `laplace_step_inner()` and `laplace_step_border()`. The first one can be called before all the communications have finished and the second one needs to be executed only with the interchanging data completed because of the data dependency.

```
  while ( gerror > tol*tol && iter < iter_max )
  {
    if (me > 0) {
        MPI_Isend(A + R0, n, MPI_FLOAT, me - 1, 33, MPI_COMM_WORLD, &request0);
        MPI_Irecv(A, n, MPI_FLOAT, me - 1, 33, MPI_COMM_WORLD, &request1);
    }
    if (me < nprocs - 1) {
        MPI_Isend(A + RF, n, MPI_FLOAT, me + 1, 33, MPI_COMM_WORLD, &request2);
        MPI_Irecv(A + RF + n, n, MPI_FLOAT, me + 1, 33, MPI_COMM_WORLD, &request3);
    }
    iter++;
    error = laplace_step_inner (A, temp, n /*col*/, me, nprocs);
    if ( me ) MPI_Wait(&request1, &status);
    if ( me != nprocs - 1 ) MPI_Wait(&request3, &status);
    error1 = laplace_step_border (A, temp, n /*col*/, me, nprocs);
    error = error > error1 ? error : error1;
    MPI_Allreduce(&error, &gerror, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
    float *swap = A; A = temp; temp = swap; // swap pointers A & temp
  }
```

In this new version, we use new communication functions that contrary to the old ones do not block the code until the communications are done [2]. As we can see, the `laplace_step_inner()` is called before the receiving request is completed. This function is the same as the `laplace_step()` from the MPI base implementation but now the computations are performed from the row 3 to the n/N whic are the ones that do not depend on other data from different partitions.

```
float laplace_step_inner(float *in, float *out, int n/*col*/,
                   int me, int nprocs)
{
  int i, j;
  float error=0.0f;
  for ( j = 2; j < n/nprocs; j++ ) /* rows */
    for ( i = 1; i < n - 1; i++ ) /* cols */
    {
      out[j*n+i]= stencil(in[j*n+i+1], in[j*n+i-1], in[(j-1)*n+i], in[(j+1)*n+i]);
      error = max_error( error, out[j*n+i], in[j*n+i] );
    }
  return error;
}
```

Finally, we present the `laplace_step_border()` function that only needs to computes at maximum two rows of the partition corresponding to the borders. In this way, we need to identify again which is the number of the process in order to avoid computing the borders of the full matrix A.

```
float laplace_step_border(float *in, float *out, int n/*col*/,
                  int me, int nprocs)
{
  int i, j;
  float error = 0.0f;
  if (!me) {
    j = 1;
    for ( i = 1; i < n - 1; i++ ) /* cols */
    {
      out[j*n+i]= stencil(in[j*n+i+1], in[j*n+i-1], in[(j-1)*n+i], in[(j+1)*n+i]);
      error = max_error( error, out[j*n+i], in[j*n+i] );
    }
  }
  if (me != n/nprocs - 1) {
    j = n/nprocs;
    for ( i = 1; i < n - 1; i++ ) /* cols */
    {
      out[j*n+i]= stencil(in[j*n+i+1], in[j*n+i-1], in[(j-1)*n+i], in[(j+1)*n+i]);
      error = max_error( error, out[j*n+i], in[j*n+i] );
    }
  }
  return error;
}
```

We will refer to this version of the code as *Version 2*.

Hybrid solution Lastly, the code can be optimized by using OpenMP in each one of the processes to execute threads over the available cores. This optimization needs to take care of the number of the cores of each node and the number of the threads that each core will execute in order to not use more resources than the available ones.

To implement this optimization we recover the MPI base implementation and then we execute the threads in the `laplace_step()` function considering that we need to take the maximum error among all the threads of the given process. In this way, we will be executing and killing the threads in each iteration of the laplace equation.

```
float laplace_step(float *in, float *out, int n/*col*/,
                  int me, int nprocs)
{
  int i, j;
  float error=0.0f;
  int i_first = 1, i_last = n/nprocs + 1;
  if (me == 0) i_first++;
  if (me == nprocs - 1) i_last--;
#define N n/nprocs + 2 /* Number of rows of the new matrix */
#pragma omp parallel default(shared)
  {
  threads = omp_get_num_threads();
#pragma omp for private(i,j) reduction(max:error)
  for ( j = i_first; j < i_last; j++ ) /* rows */
    for ( i = 1; i < n - 1; i++ ) /* cols */
    {
      out[j*n+i]= stencil(in[j*n+i+1], in[j*n+i-1], in[(j-1)*n+i], in[(j+1)*n+i]);
      error = max_error( error, out[j*n+i], in[j*n+i] );
    }
  }
  return error;
}
```

This version of the code is called through all this project as *Version 4*.

This version can be slightly modified to avoid starting and killing threads in each iteration step. In this way, we need to initialize the parallel region just before the while loop. This version needs to pay great attention to some instructions as now we are dealing with two different levels of parallelization. In this sense, we need to execute the interchanging communications by just one determined thread. In this case, for simplicity, we choose the master thread. Likewise, the reduction operation among the processes needs to be called by just one thread as well. All this steps have to be done once every thread of each process have finished. That is why we need to implement barriers before calling the master thread [3].

```
#pragma omp parallel default(shared)\
  firstprivate(iter,A,temp)
  {
  threads = omp_get_num_threads();
  float gerror = 1.0f;
  fprintf(stderr, "lap(): rank : %d thread : %d init bucle\n", me,
      omp_get_thread_num());
  while ( gerror > tol*tol && iter < iter_max )
  {
#pragma omp barrier
#pragma omp master
```

```
    {
    if (me < nprocs - 1) {
        MPI_Send(A + RF, n, MPI_FLOAT, me + 1, 33, MPI_COMM_WORLD);
    }
    if (me > 0) {
        MPI_Recv(A, n, MPI_FLOAT, me - 1, 33, MPI_COMM_WORLD, &status);
    }
    if (me > 0) {
        MPI_Send(A + R0, n, MPI_FLOAT, me - 1, 33, MPI_COMM_WORLD);
    }
    if (me < nprocs - 1) {
        MPI_Recv(A + RF + n, n, MPI_FLOAT, me + 1, 33, MPI_COMM_WORLD, &status);
    }
    }
    iter++;
    error_t = laplace_step (A, temp, n /*col*/, me, nprocs);
#pragma omp barrier
#pragma omp master
    {
    MPI_Allreduce(&error_t, &gerror, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
    }
    float *swap = A; A = temp; temp = swap; // swap pointers A & temp
  }
```

Finally, we execute the threads in the `laplace_step()` function. We use them to calculate the main loop and it is needed to set the reduction clause to obtain the maximum error among the threads for a given process. This error has to be declared as a global variable because we need to set it as a shared one if we want to perform the reduction operation.

```
  {
  threads = omp_get_num_threads();
#pragma omp for private(i,j) reduction(max:error)
  for ( j = i_first; j < i_last; j++ ) /* rows */
    for ( i = 1; i < n - 1; i++ ) /* cols */
    {
      out[j*n+i]= stencil(in[j*n+i+1], in[j*n+i-1], in[(j-1)*n+i], in[(j+1)*n+i]);
```

This version is named in this project as *Version 3*.

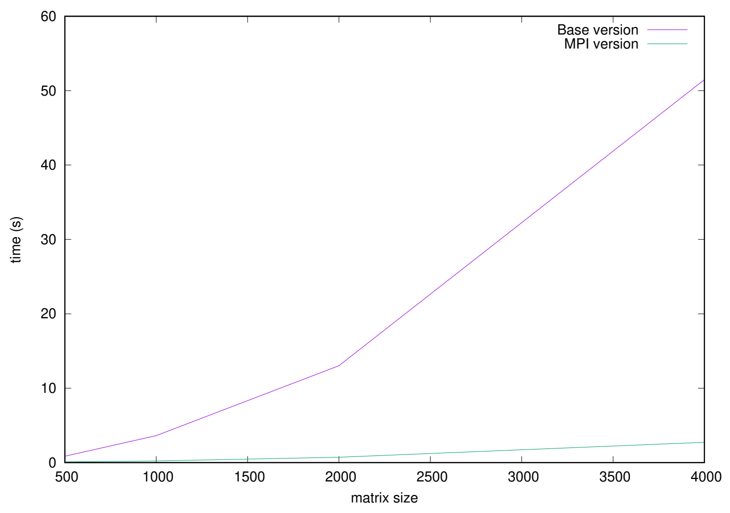


Figure 1: Performance comparison between the Base version of the code and the MPI implementation with 20 processes for different matrix sizes.

Analysis of performance results

Performance results of both the parallelized and optimized code were extracted and will be analysed in the following section with the help of some images.

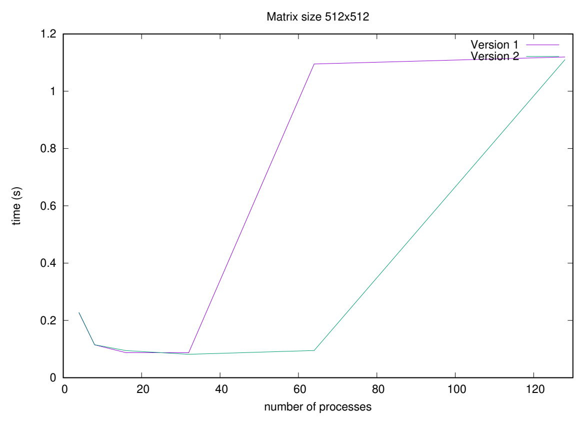
MPI Implementation

As can be seen on Figure 1, implementing MPI in the base code decreases the runtime of the problem significantly. The effect of the parallelization increases for larger matrices, with a factor bigger than 10 for large matrix sizes. This implementation allows us to compute very fast the laplace solution for large matrix compared to the old version. The figure is displayed for a fixed number of resources, in this case we are using only 20 cores which means that we expect to enhance this difference with the increasing of the resources.

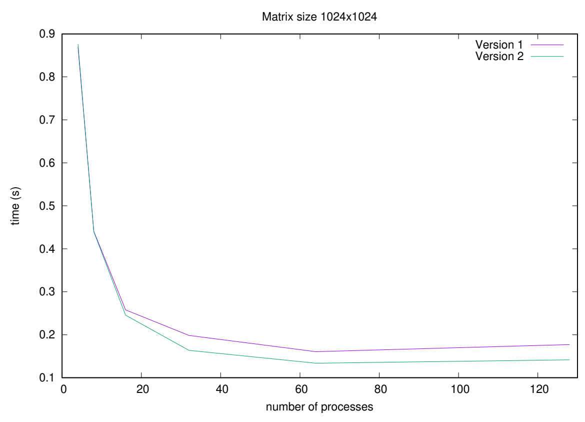
Optimization by Overlapping communication and computation

After implementing the first optimization, results were extracted for fixed matrix sizes and increasing number of processes. The result are shown on Figure 2 and it can be concluded that this optimization decreases runtime, with a factor that is larger for bigger matrix size. This is, as explained before, because a process can start computing the inner points of a matrix at any given time, independent of the data received from other processes. In addition, we can observe how a performance limit exists as there is a certain number of processes for which we can not decrease the elapsed time. Note that the Figure 2a shows very weird results as the performance time is not big enough to hide the fluctuations.

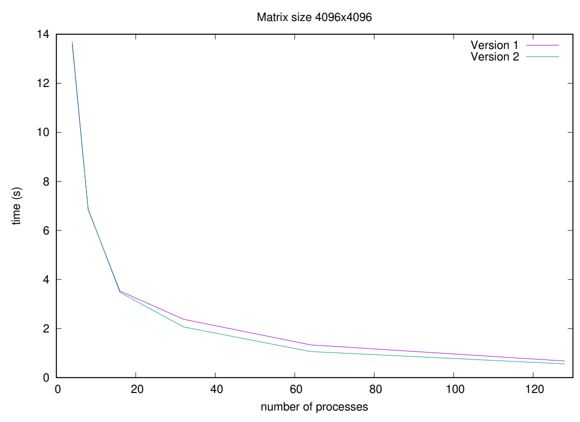
Figure 3 shows the results for runtime when the amount of cores is fixed and the matrix size increases.



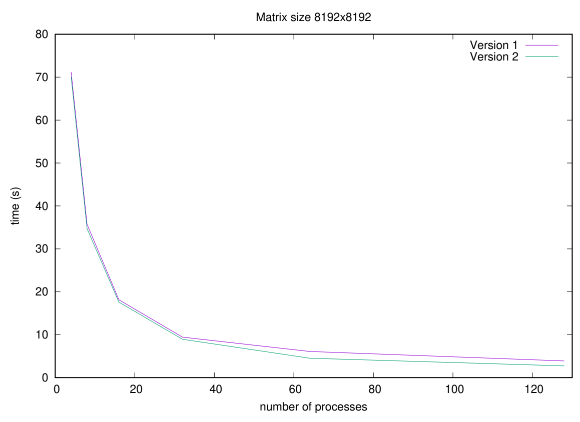
(a) Matrix size 512 × 512.



(b) Matrix size 1024 × 1024.



(c) Matrix size 4096 × 4096.



(d) Matrix size 8192 × 8192.

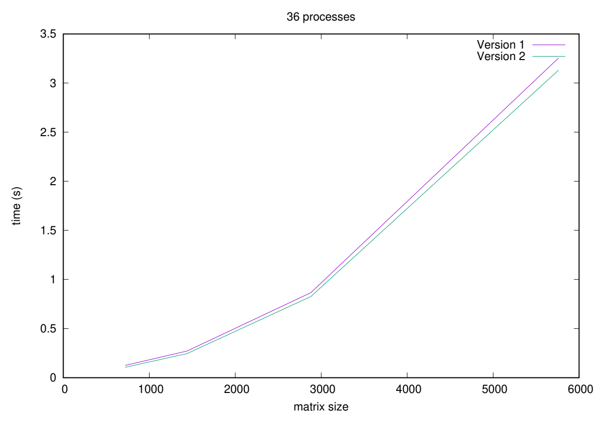
Figure 2: Comparison of the performance results between the MPI base version and the optimization version 1 for a fixed matrix size.

The results are again slightly better for the optimized code than for the simple MPI implementation. Even though, we expect to observe a slightly decrease on the observed slope of the line when increasing more the size of the computed matrix. This is because the communication costs increase with the size of the matrix and some optimizations are lost.

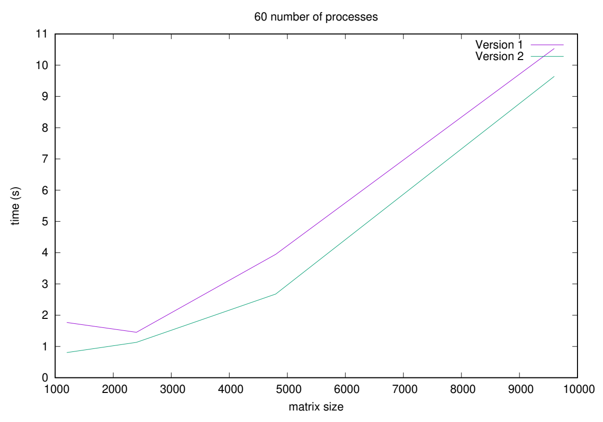
Figure 4 shows results when both matrix size and number of resources vary. We are multiplying by two both the number of resources and the number of stencil operations so we expect that the time is constant. This is in practice not the case as it is seen in the mentioned figure due to the increasing cost of communication when increasing matrix size or resources. Note that the size is multiplied by two so the number of operations is really multiplied by 4.

Hybrid solution

After implementing both the base MPI version and first optimization, we make it with the two hybrid optimizations. The performance of this last version is compared with the base MPI implementation and the results of Figure 5 were obtained. It shows a comparison between the base version of the MPI and



(a) 36 number of cores.



(b) 60 number of cores.

Figure 3: Comparison of the performance results between the MPI base version and the optimization version 1 for a fixed number of resources.

the two hybrid version with the same amount of resources (120) and varying the number of threads that extend each one of the processes for the hybrid version. Implementation of the hybrid solution results in lower runtime than for the original MPI solution, but when accessing a high amount of threads (12), results do not improve a lot, especially for large matrix sizes. The optimal amount of threads is six which corresponds to 2 processes per node and 6 CPU's per task.

Lastly, Figure 6 shows a comparison in runtime between the two hybrid versions for the optimal number of threads and 120 resources. Version 4 shows slightly better results. This could be because the cost of interchanging information is more important than the handling of the threads. Even though, as the size matrix increase, both versions display similar results.

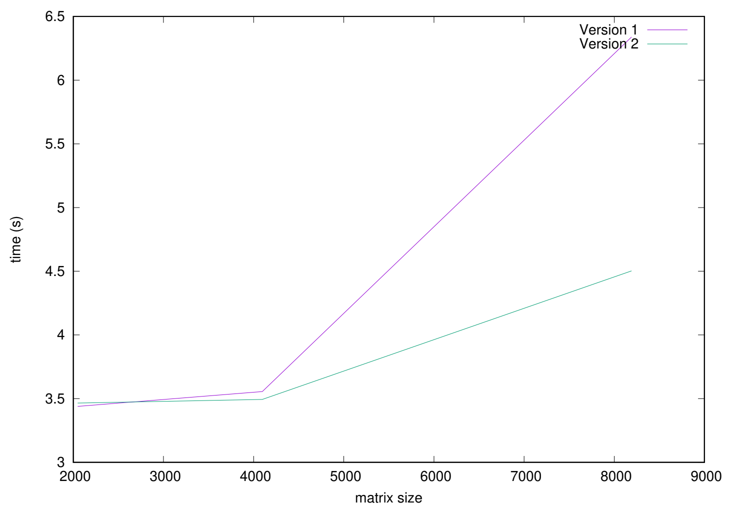
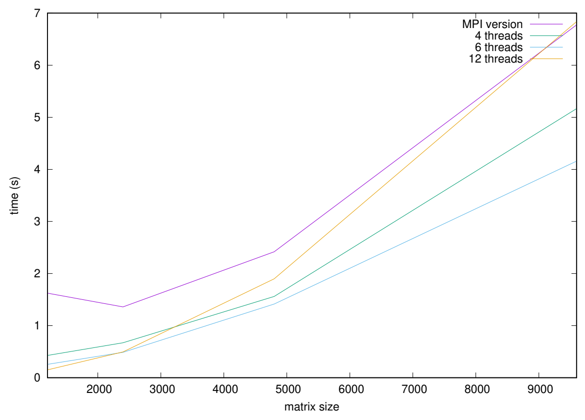
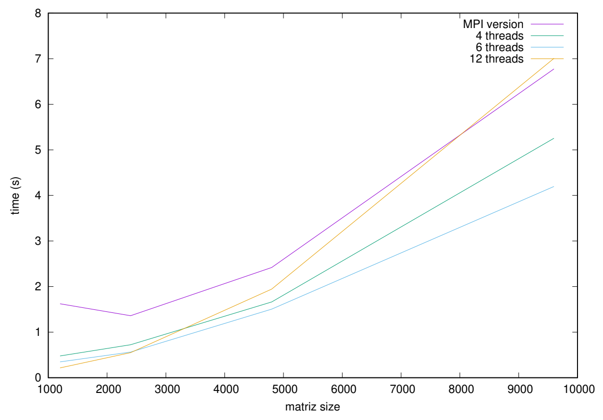


Figure 4: Comparison of the performance results between the MPI base version and the optimization version 1 and 2 for variable dimensions. The matrix size is $1024 \cdot 2^i$ and the number of resources are 4^i for $i = 1 \ldots 3$.



(a) Version 04



(b) Version 03

Figure 5: Performance comparison between the MPI base implementation and the hybrid optimization versions for a 120 number of resources and different matrix sizes and threads.

Conclusion

Both the implementation of MPI and the different optimizations decreased the runtime of the problem. This result was more significant for matrices with large size. On the other side, cost of communication needs to be taken into account as it influences the runtime when the matrix size or the amount of resources increase, and can cancel out the original improvement obtained by the optimizations. In addition, we have seen how it works a version implementing the two known parallelization methods. As we observed, this version yields small improvements regarding the base MPI implementation but it also introduces a new tunable parameter which is the number of CPU's per task, the number of threads. This is a variable that is difficult to select his optimal parameter.

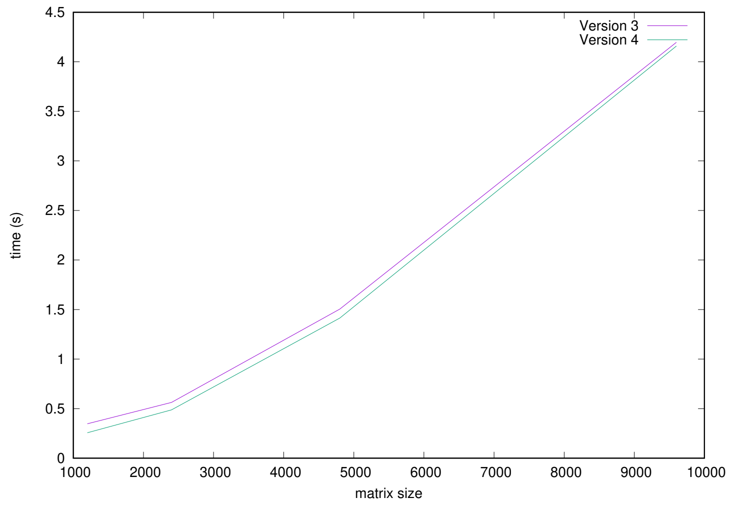


Figure 6: Performance comparison between the two hybrids versions for 120 resources distributed in 6 threads per process for different size problems.

References

[1] Valeria Cardellini, Alessandro Fanfarillo, and Salvatore Filippone. “Overlapping Communication with Computation in MPI Applications”. In: February (2016), pp. 1–19.

[2] Eduardo César. *Parallel Programming with MPI*. 2018.

[3] Rolf Rabenseifner et al. “Hybrid MPI and OpenMP parallel programming”. In: *Lecture Notes in Computer Science*. Vol. 4192 LNCS. 2006, p. 11. ISBN: 354039110X.