
Programming of Supercomputers WS 12/13

Final report

Marco Seravalli

January 24th 2013

1 Introduction

The Programming of Supercomputers laboratory is about understanding how sequential applications can be modified and optimized in order to be executed efficiently and effectively on massively parallel machines.

The laboratory is divided into two main parts: the first one is about understanding how a sequential code can be optimized by acting on the input phase and on the compilation phase.

The second part is instead more focused on parallelizing a sequential Computational Fluid Dynamic application. This process is performed throughout four steps (the milestones in our case).

- In the first phase the domain has to be decomposed, better if using different strategies. First of all a straightforward approach was adopted: in this case contiguous blocks of elements are assigned to a single processor without taking the actual position in space into account. Afterwards also some advanced partitioning algorithms provided by the `metis` library were introduced namely `metis-dual` and `metis-nodal`.
- The second stage consists in building the communication lists that have to be used for the successive computation part. In this phase also other positioning arrays have to be adapted in order to respect the correctness of the communication.
- The third step involves the actual parallelization of the computational loop. This requires to understand how the data within the loop will be exchanged during the computation.

- The aim of the fourth and last part is the optimization of the code in order to be able to deliver a better performing application that satisfies some predefined standards. The particular objective in this case is to reduce running time and to have a communication overhead for a specific problem under the threshold of 25%.

2 Sequential optimization

The sequential optimization consisted in applying different compiler flags at compilation time in order to see the effects produced by those in terms of speedup, cache hit improvements and floating point operations per second (flops). Then the optimizations were validated by running the application with different input files. In following graphs it is possible to see how the various compiler optimization affect the execution time (Figure 1) and the L2 cache miss rate (Figure 2).

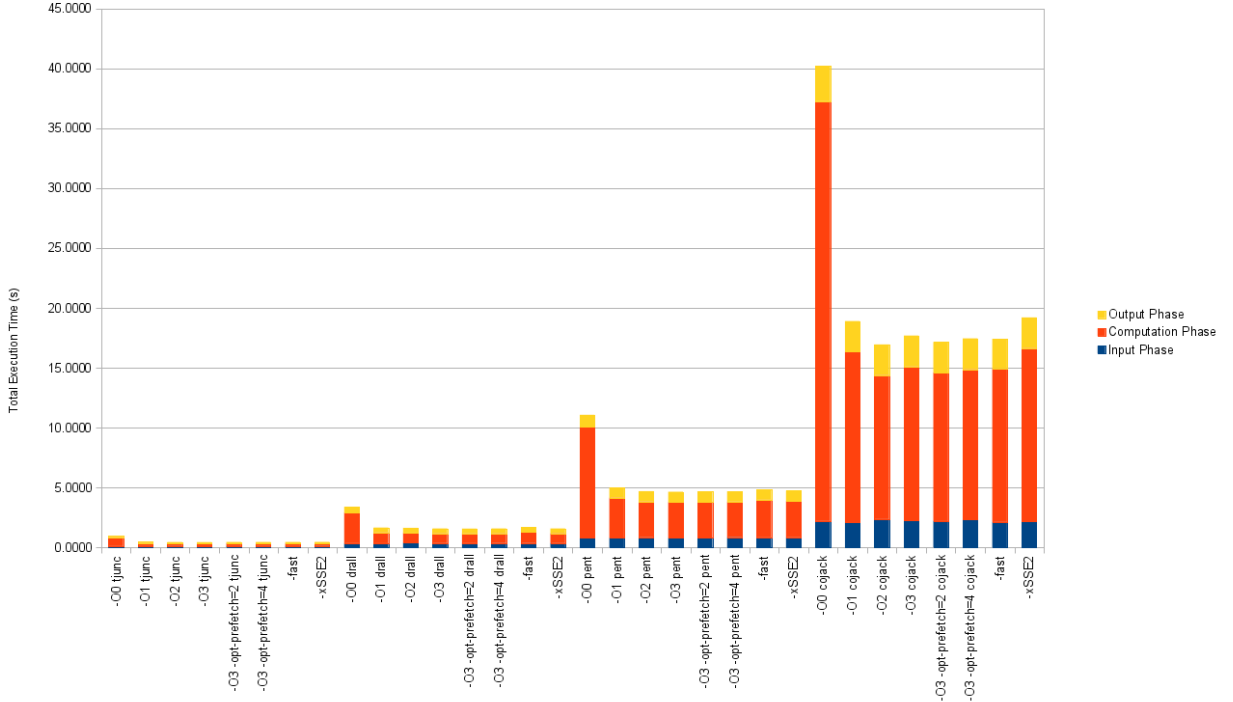


Figure 1: Execution time for each phase.

Furthermore, also the file reading phase is modified from text to binary. This means that the input files are converted from ASCII values to a binary representation. Also this change provides good speed enhancements in the file reading phase. The improvement can be quantified in about one order of magnitude. In Figure 3 we can see the runtime improvement and how this is related to the size of the input file.

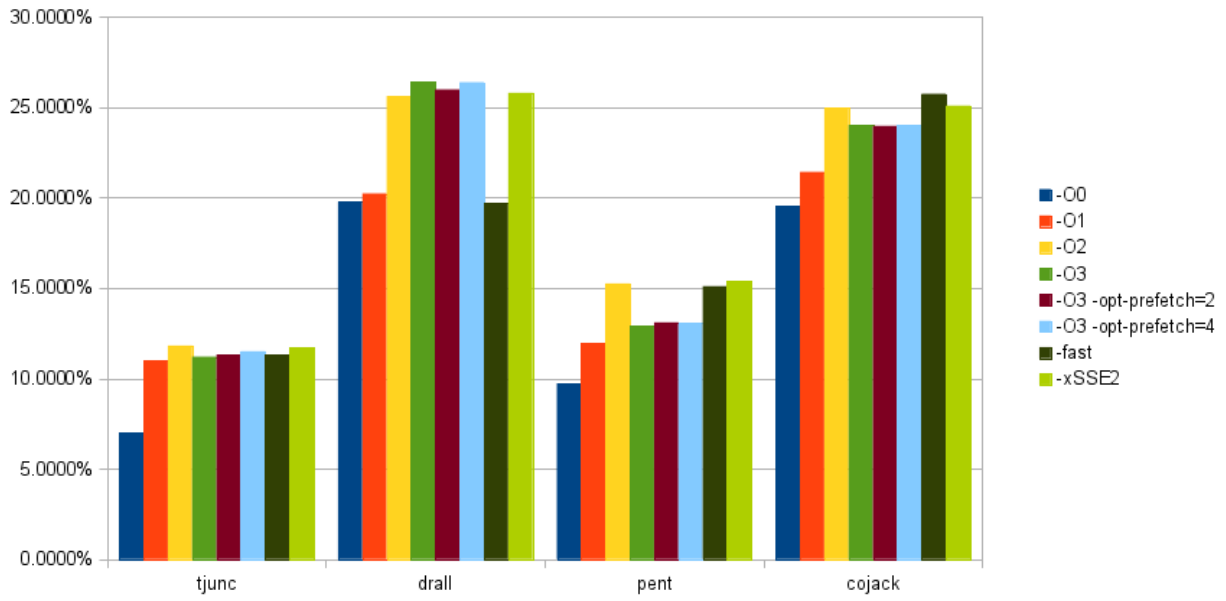


Figure 2: L2 cache miss rate.

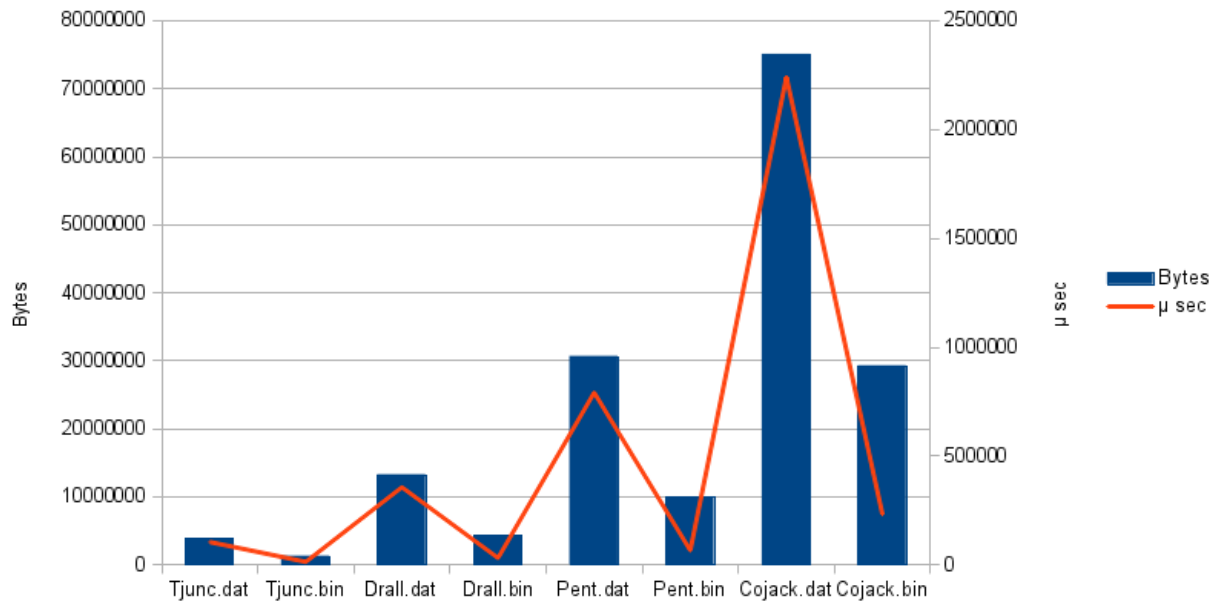


Figure 3: Execution time of the input phase in relation to the input file size.

3 Benchmark parallelization

Data distribution

The data are distributed using different partitioning schemes. In particular the classical distribution and two different distributions offered by the **Metis** partitioning library.

For the classical distribution the partitioning is performed by splitting the domain composed of n cells equally among the computing units (np) in a sequential way, i.e. the first $\frac{n}{np}$ cells are assigned to the first process, then the second set of $\frac{n}{np}$ sequential cells is assigned to the second process and so forth.

For the two other distributions the whole array containing the cells is passed to the functions provided by the **Metis** library. For more information about how the distribution is performed by these functions please refer to the Metis Homepage.

The partitioning was firstly performed on a single processor, eventually in order to decrease the communication among the computing units, this step was computed in parallel by every single processor.

The different partitioning strategies are handled by two different functions and for these the code duplication is reduced to the minimum: the partitioning functions only perform some strictly necessary tasks. In order to achieve this result as much code as possible was shifted in the calling function.

Communication model

After the partition the cells are processed by different computing units and it is then necessary to be able to communicate these values among the different processes. For the communication model send and receive lists have been used. In particular it has been necessary to guarantee that the received data was actually received at the correct position. The communication lists specify which of the internal cells have to be sent and which of the cells have to be received. The received cells are not part of the internal domain assigned to every processor and from now on they will be referred as ghost cells. Additional lists containing the number of elements needed to be sent and received from other processors were also introduced.

Once the communications lists are created the global to local index can be created from those. The idea for this index is first to see what elements belong to a specific processor and assign them an incremental number. Then during a successive traversal the ghost cells are incrementally numbered. The actual implementation can be seen in the following code.

```
// initialise global_local
int count = 0;
for (int i = 0; i < el_int_glob; ++i) {
    if (part_elems[i] == my_rank) {
        (*global_local)[i] = count;
        ++count;
    }
}
```

```

    } else {
        (*global_local)[i] = -1;
    }
}

for (int i = 0; i < size; ++i) {
    for (int j = 0; j < recv_count[i]; ++j) {
        if ((*global_local)[recv_list[i][j]] == -1) {
            (*global_local)[recv_list[i][j]] = count;
            ++count;
        }
    }
}
}

```

The reason for such numbering is mainly due to the successive phase: the building of the local to global index. In this index, we want to have the inner elements at the beginning of the array and the ghost cells at the end. Since the global to local index holds that specific structure it is possible to initialize `local_global` in a single traversal as it can be seen in the following code snippet.

```

// after the initialization of global_local we build up local_global
for (int i = 0; i < el_int_glob; ++i) {
    if (global_local[i] != -1) {
        (*local_global)[global_local[i]] = i;
    }
}
}

```

After having the indexes set it is also possible to modify the communication lists and the `lcc` array in order to have the local indexes of the elements. This allows to apply only a minimum amount of changes in the original code. A further issue that had to be solved, regards the external cells, i.e. the cells outside the boundaries. The applications treats all these cells in the very same way and their coefficients are always 0. It follows that all the information regarding these elements can be stored in a single position. This is performed by making the coefficient arrays one element longer and storing this additional information in the last cell. Of course also `lcc` needs to be modified accordingly, such that all the external cells point to the very same position in the coefficient arrays. This change yields to the final form of the `local_global_index` that can be seen in Figure 4.

MPI implementation

For the MPI implementation different aspects should be taken into account:

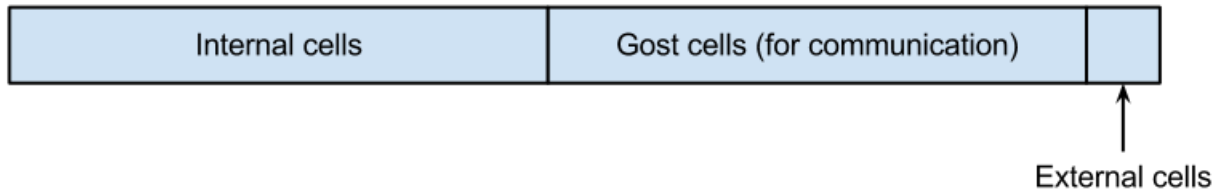


Figure 4: How elements are stored within the local to global index.

- Point to point operations: the exchange of the cells among the different processors is handled by point to point operations. In such scenario every process sends the information only to another processor at a time.
- Non blocking communication: if blocking point to point communication is used, deadlocks might occur. The reason is that the some processes need to send data before being able to receive information. If it happens that all processes need to communicate with all processes, it might occur the case that all processes are ready to send, but none is ready to receive, causing then a deadlock. This can be avoided by using non blocking communication. The sending function returns immediately even if the data are not actually send. Also for the receiving non blocking communication can be exploited. In this case the advantage would be that the incoming data from all processes can be received in the background. Eventually, before performing further operations on the received and sent data there is the necessity of waiting for the communication to be completed.
- Indexed data: for sending data indexed data types are adopted. With this data type model MPI sends only the necessary elements from the correct position of each array. When receiving the data another indexed data type should be used instead, because the data has to be received in some other positions of the array. In the current implementation, there are created send and receive data types for every process with whom there is a communication. The building of these indexes is performed by using the send and receive lists previously created.

```
for (int i = 0; i < size; ++i) {
    if (send_count[i] > 0) {
        block_len = (int*) calloc(send_count[i], sizeof(int));
        for (int j = 0; j < send_count[i]; ++j) {
            block_len[j] = 1;
        }
        MPI_Type_indexed(send_count[i],
                        block_len,
                        send_list[i],
                        MPI_DOUBLE,
                        &(send_types[i]));
    }
}
```

```

        MPI_Type_commit(&(send_types[i]));
        free(block_len);
    }
    if (recv_count[i] > 0) {
        block_len = (int*) calloc(recv_count[i], sizeof(int));
        for (int j = 0; j < recv_count[i]; ++j) {
            block_len[j] = 1;
        }
        MPI_Type_indexed(recv_count[i],
                        block_len,
                        recv_list[i],
                        MPI_DOUBLE,
                        &(recv_types[i]));
        MPI_Type_commit(&(recv_types[i]));
        free(block_len);
    }
}

```

- Collective operations: some of the variables of the serial implementation are computed by using all elements of the domain. Of course the parallel code needs to take into account this computations and one of the most efficient ways to compute these values is via collective operations. In particular for the given implementation `MPI_Allreduce` was adopted. This function allows to all the processes to share the result once it is computed, hence afterwards no further communication is required.

4 Performance analysis and tuning

For this last phase first of all an optimization aim is set, then the code is analysed using different profiling tools and after the analysis some modifications are performed in order to reach the predefined objective.

Performance aims

The major objective of the performance is to reduce the total execution time of the application with a particular focus on the communication overhead, which should be under 25% for the whole software in the case of using 4 processes.

Profiling

The code is analyzed using different tools, namely the ScoreP framework in collaboration with the PAPI library and the Periscope tool. ScoreP in particular was used for instrumenting the code and then manually inspect the outcome through Cube, a visualization

tool that allows to scan the different dimensions of the profiling result. Periscope was instead employed to automatically find bottlenecks and hot-spots of the application.

The outcome of the first analysis is that there is particularly high communication overhead in the initialization part. This is due to the fact that the partitioning is performed on a process and then distributed to the others.

The computational loop presents some overhead only in the **Allreduce** MPI function. Small overhead is due to the communication functions **send** and **recv** because asynchronous communication is adopted since the first parallelization.

The finalization phase does not present a great overhead because all the data are communicated only once and the largest amount of time is spent in writing information on the disk.

Optimization

The communication overhead in the first part could be overcome by computing directly the partitioning on all processes in order to minimize communication. Even though in this case more computations are performed, the overall performance is not affected.

To reduce the communication overhead, all the duplicates from the communication lists are deleted so that only the needed information was exchanged. Unfortunately, this strategy produces a soar in the initialization phase, but a better performance of the computational loop.

Furthermore in order to be able to overlap communication and computation the sending and the receiving of data was modified after the initial implementation. In the first approach the data was communicated at the beginning of the computational cycle using **IRecv** and **ISend** in that order.

```
while ( iter < max_iters ) {
    update the old direc1 values
    receive direc1 values
    send direc1 values
    ...

    update resvec
    ...
}
```

However in order to be able to perform computations while communicating, calculation and the sending of **direc1** was moved after the computation of **resvec**. This required to rearrange also the first send and the last receive statement as it can be seen in the following pseudo code.

```
update the old direc1 values
send direc1 values
```



```

while ( iter < max_iters ) {
    receive direc1 values
    ...

    update resvec
    update the old direc1 values
    send direc1 values
    ...
}

receive direc1 values

```

Unfortunately, the improvements yield by this optimization step are not directly visible. One of the possible causes could be the fact that also the reference code contains asynchronous communication and the optimisations performed to not lead to significant changes in the adopted concepts.

Moreover, also the bottleneck caused by the **Allreduce** statements cannot be further optimized. The reason lies on the sequential nature of the problem where the next computations rely on the previous ones. In addition to that also non blocking operations cannot be introduced because there is no possible overlap between the communication and the computation: the calculated data have to be immediately sent to the neighbours to calculate the next coefficient.

Experiment results

The experiments were run on the interactive partition of the Linux Cluster, the specifics of the machine adopted can be found under lrz.de. The application was run using two different set of input namely **pent** and **cojack** using different partitioning algorithms (classical and Metis dual). Every configuration had been run on an increasing number of processors, from 1 to 64.

In figures 5 and 6: it can be seen the different execution times.

The particularly high cost of the initialization is due to the fact that the duplicates are removed send and receive lists, in order to decrease the amount of communication needed. The drawback of this procedure is that the lists have to be sorted, which causes the overhead of the initialization part.

In the context of this laboratory, the proposed solution would be suboptimal because the initialization cost results to be higher than the actual computation cycle. Anyway, for applications that would involve more time steps, so that the computational loop is more consistent in terms of computing power demands, the proposed approach might result to be a better solution. Unfortunately, these tests and a more careful and deep comparison with a code containing duplicates in the lists could not be performed due to the unavailability of the batch and interactive servers.

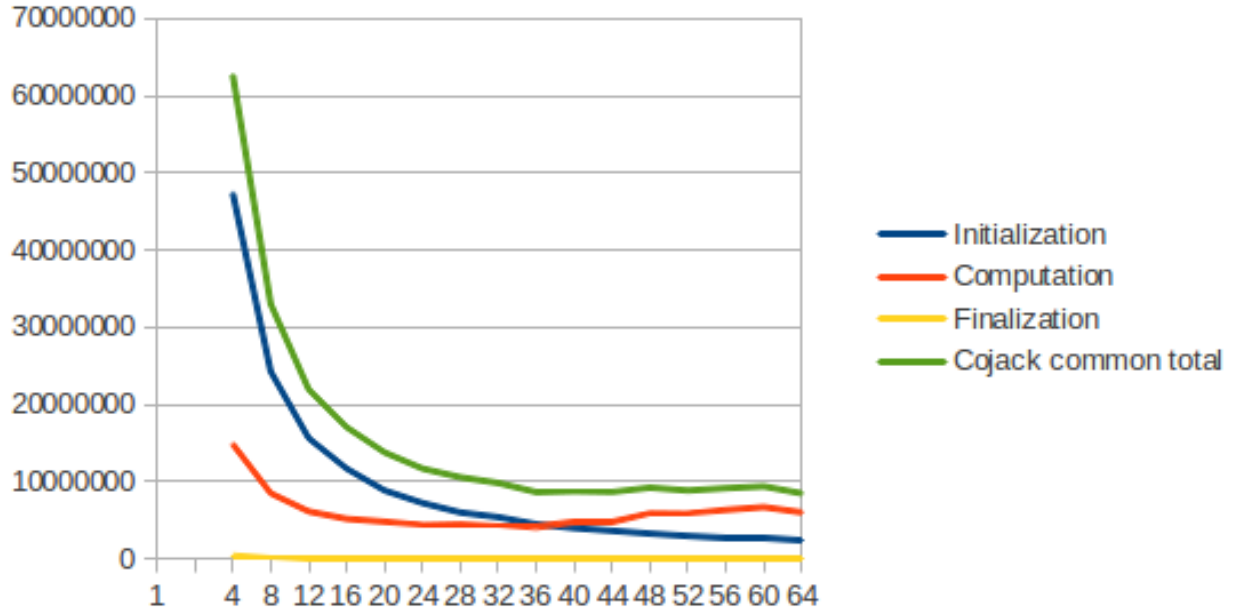


Figure 5: Times in μsec of the single parts of Cojack input run on multiple processes using the common partitioning algorithm.

In any case, the speedup obtained for the computational cycle presents a good scaling, in particular for small number of processors as it can be seen in figures 7 and 8.

For high number of processors the application does not benefit from the parallelization any more, because of the increased communication overhead.

The results for the **pent** geometry resemble the ones obtained for **cojack**. This data including additional charts regarding all the configurations can be found in the spreadsheet **profiling.ods**.

Eventually also the communication objective has been fulfilled, both in **pent** and in **cojack** the percentage of time spent in communication is less than 20%, when 4 processes are used.

5 Overview

The lab course proposed an opportunity to approach the parallelization of an existing code. During the different phases several unexpected challenges had to be solved. In particular, a careful planning of the data structures and the communication had been necessary to be able to reach a working solution.

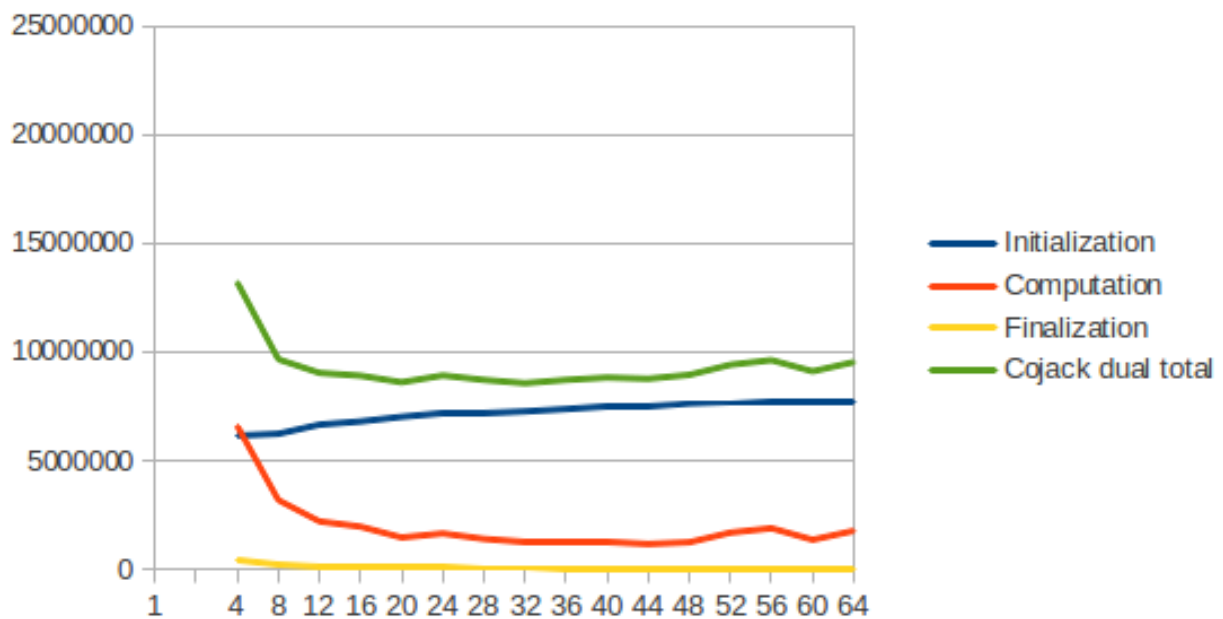


Figure 6: Times in μsec of the single parts of Cojack input run on multiple processes using the dual partitioning algorithm.

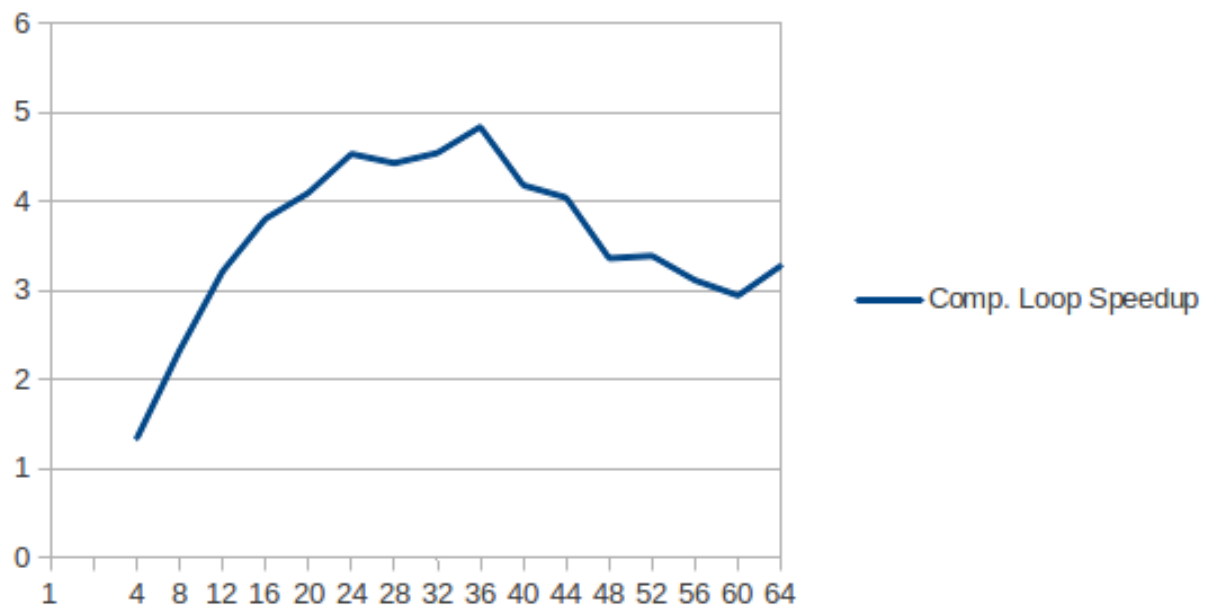


Figure 7: Scaling for computational loop for Cojack input run on multiple processes using the common partitioning algorithm.

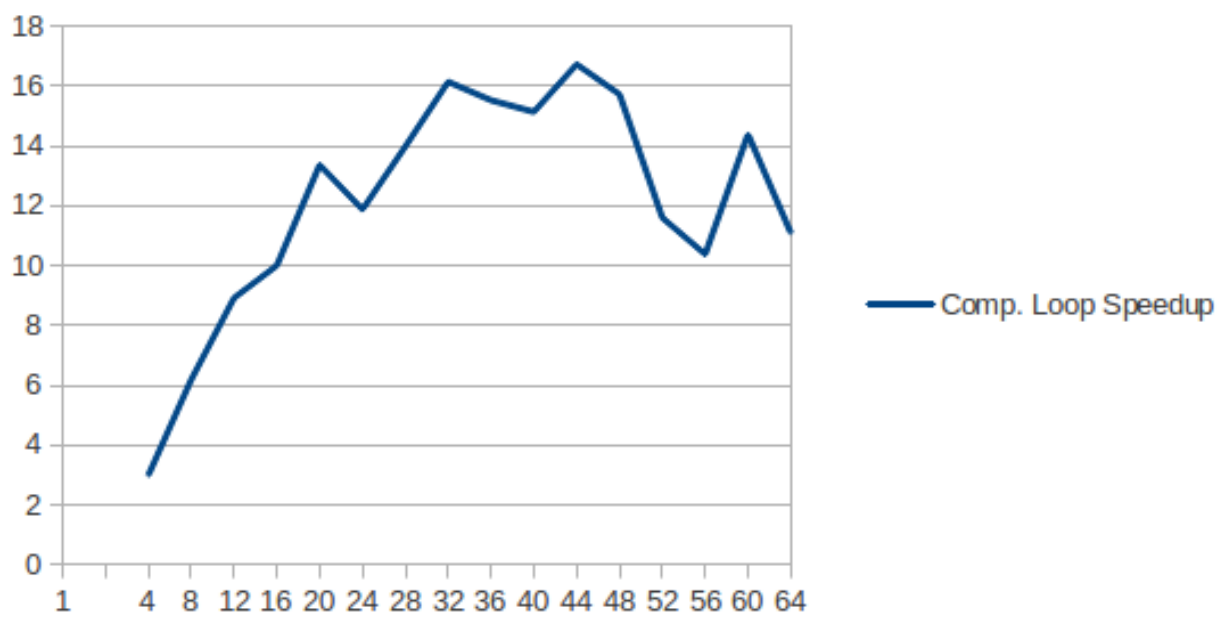


Figure 8: Scaling for computational loop for Cojack input run on multiple processes using the dual partitioning algorithm.