Programming of Supercomputers

Final report

Miklós Homolya, Tushar Upadhyay

February 1, 2014

1 Introduction

Our job throughout the course was to optimize, parallelize and tune a generalized orthomin solver from the Fire benchmark suite. The work was divided into two assignments: first the sequential optimization, and then the MPI parallelization of the program.

The benchmark program uses an unstructured mesh of volume cells for domain discretization. Neighboring information is stored explicitly, and thus neighbors are accessed through indirect addressing. The algorithm iterates until acceptable residual is achieved.

Both the measurements of the sequential optimization and the MPI parallelization of the benchmark program were carried out on the SuperMUC supercomputing cluster.

2 Sequential optimization

During the performance optimization of an application, the first thing should always be the optimization of serial performance. Serial optimization comes with no communication overhead, and usually also reduces energy consumption. Parallelization, on the other hand, intends to reduce execution time, but it often increases energy consumption due to communication and parallelization overhead.

One of the most important observations of the first assignment was that the generalized orthomin solver is rather memory-bound than computation-bound. We concluded this from the low utilization of floating-point units and the high cache miss rates.

During the analysis, we relied on compiler optimizations enabled/disabled by compiler flags. We compared the following optimization levels:

- -g
- -O1

- -O2
- -O3
- -O3 -opt-prefetch

A comparison of the execution times of the computation phase with different optimization levels is visible on Figure 1. The best performance is achieved at optimization level `-O2`. We compared only the computation phase on the chart, because the I/O phases are not really affected by compiler optimization flags.

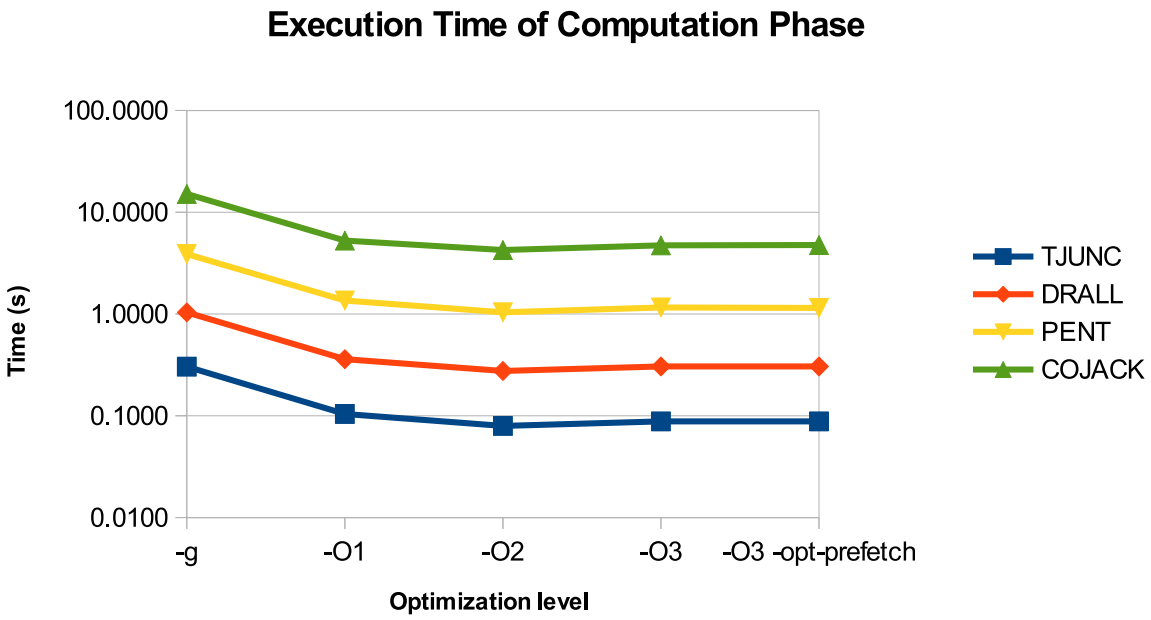


Figure 1: Comparison of different optimization levels.

The I/O phases are responsible for significant portion of the execution time, especially at high optimization levels, small datasets, and textual input/output files. An example for the `cojack` dataset at optimization level `-O2` can be seen on Figure 2. Usage of binary input format significantly reduces the execution time of the INPUT phase. Note that the `.vtk` files are also textual, so we expect similar performance improvement of the OUTPUT phase using a binary output format.

3 Benchmark parallelization

Large supercomputer clusters, such as SuperMUC, have numerous computational nodes, each with its own memory. We distribute the computational domain among the processes, and use MPI for communication. To update each volume element, we need data from

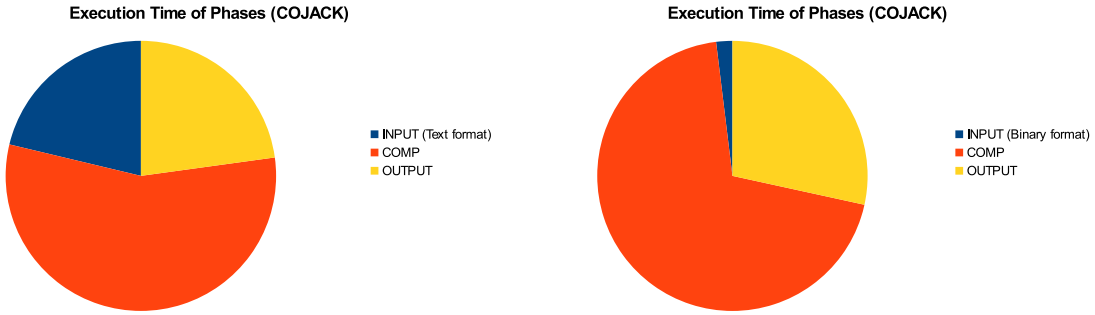


Figure 2: Contribution of different phases to the execution time.

its neighboring elements only. We use communication to exchange these values between processes.

3.1 Data distribution

For the Milestone 1, we had to implement three ways of distributing the volume elements among processes:

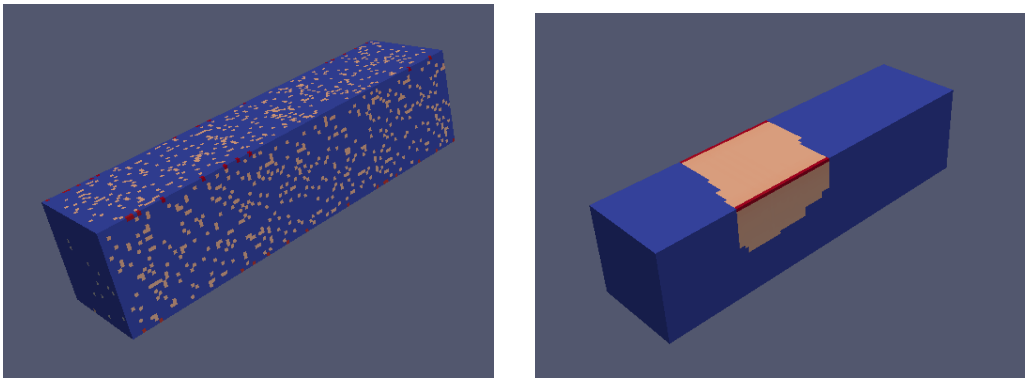
- **Classical**, where volume elements were distributed based on their position in the input file.
- **METIS dual**, using `METIS_PartMeshDual` from the METIS library for multilevel graph partitioning, where the volume elements become the vertices of the graph.
- **METIS nodal**, using `METIS_PartMeshNodal` from the METIS library for multilevel graph partitioning, where nodes become the vertices of the graph.

The METIS library has two versions, which we could have used. They are the serial version of METIS, and ParMETIS, a parallel, MPI-based version of METIS. Since ParMETIS requires manual redistribution of data after mesh partitioning, which practically implies all-to-all communication, we decided to stick to the serial version of METIS. This made implementation easier.

We considered two approaches for initialization. Firstly, running serial METIS on one process, and then scattering relevant data to the other processes. Secondly, running serial METIS on all processes at the same time, and then discarding irrelevant data in each process. We chose the second approach, so that we spared the communication time of scattering data. However, we must note, that this approach could lead to poor energy-efficiency in case of many processes, because we run the same calculations on each process.

METIS partitioning required us to implement a general way of data distribution and global-local mapping. In case of classical partioning, we used the same general code, but instead of calling `METIS_PartMeshDual` or `METIS_PartMeshNodal`, volume elements were assigned to processes based on their indices only.

A good data distribution minimizes communication overhead. This means as few as possible cells at the boundaries, and as few neighboring subdomains as possible. The best results are achieved by METIS dual distribution, METIS nodal being slightly worse. The classical distribution gives fairly random distribution of volume elements, which causes huge communication overhead. A subdomain of the `pent` dataset is visible on Figure 3 using classical and METIS dual partitioning.



(a) Classical distribution (b) METIS dual distribution

Figure 3: Comparison of classical and METIS dual distribution.

3.2 Communication model

The serial version of the GCCG solver deals with *internal* cells, which are updated in every iteration, and *external* cells, which are not updated, but they are neighboring to internal cells. In the data distribution phase, we distribute the internal cells only. The function `compute_solution` sees only local internal cells as "internal" cells, and neighboring cells to local internal cells as "external" cells. Some of the latter are "real" external cells, which are initialized once and never updated, and some are *ghost* cells, which are internal cells of neighboring computational subdomains. Ghost cells must be updated in every iteration by means of communication between processes.

To update ghost cells, we create send and receive lists in the initialization phase. We assemble the send lists "manually" (as described in Algorithm 1), then we convert them to global indices, send them to the other processes, which then convert the global indices to their local indices, and thus they will have the receive lists. Algorithm 1 guarantees that no cell value is sent more than once to the same process.

3.3 MPI implementation

The GCCG solver never accesses ghost (and external) cells directly. They are only accessed as neighbors of internal cells, and only in case of the so-called `direc1` array. Therefore we added code to exchange the adequate values of `direc1` between processes. We start

Algorithm 1 Send list assembly

```
for all u ∈ local internal volume elements do
    N ← ∅
    for all v ∈ neighbors of u do
        if v is ghost cell then
            N ← N ∪ {owner of v}
        end if
    end for
    for all n ∈ N do
        add u to the send list for n
    end for
end for
```

all communication through non-blocking MPI routines (`MPI_Isend` and `MPI_Irecv`), and then wait for the communication to finish with `MPI_Wait`. This approach allows one-to-one communications to proceed in parallel.

The code implicitly contains several reduction operations, where values are accumulated from all elements of an array in for loops. In order to preserve the behavior of the program, we had to add reduction operations at MPI level as well. We continue to reduce local values with for loops, and then we call `MPI_Allreduce` to carry out global reduction. These were all we had to do to parallelize the computation phase.

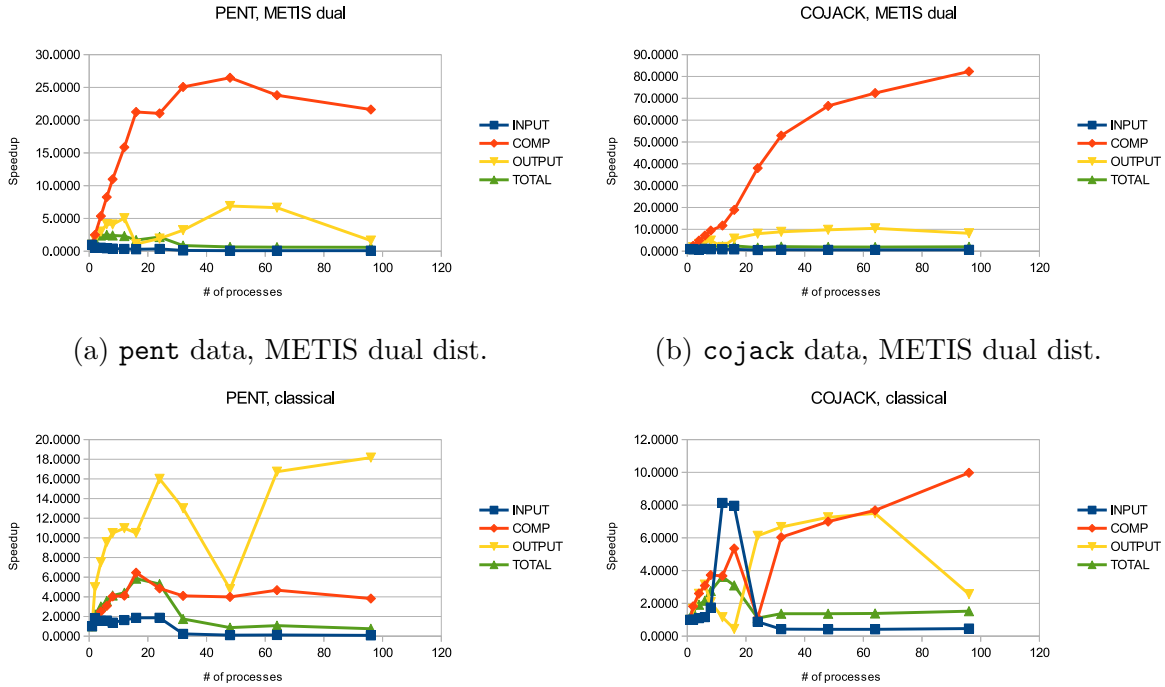
4 Performance analysis and tuning

4.1 Performance analysis

First, we gathered a lot of performance data for different data sets (`pent` and `cojack`), different data distribution methods (classical and METIS dual graph partitioning), and different number of parallel processes. We measured the execution time of initialization, computation, and finalization phases at optimization level `-O2` with `PAPI_get_real_usec` and `MPI_Barrier`. These functions are less intrusive than *Score-P* instrumentation, and thus they are likely to give more accurate performance data. The speedup graphs are presented in Figure 4.

We were surprised to see that the measured execution times of the initialization phase were fairly chaotic, even though we did every measurement 3 times, and we did not include `MPI_Init` in the initialization phase. The execution time of the initialization phase actually increases by the increase of the number of processes. For the computation phase, speedup is linear (or even superlinear) in the beginning, then it slows down and eventually declines. The finalization and output phase shows some speedup, but this speedup is much less than what we have seen for the computation phase.

Since the computation phase scales far better than the I/O phases, its share in the overall execution time becomes very small with increasing number of processes. For many



(a) `pent` data, METIS dual dist.

(b) `cojack` data, METIS dual dist.

(c) `pent` data, classical dist.

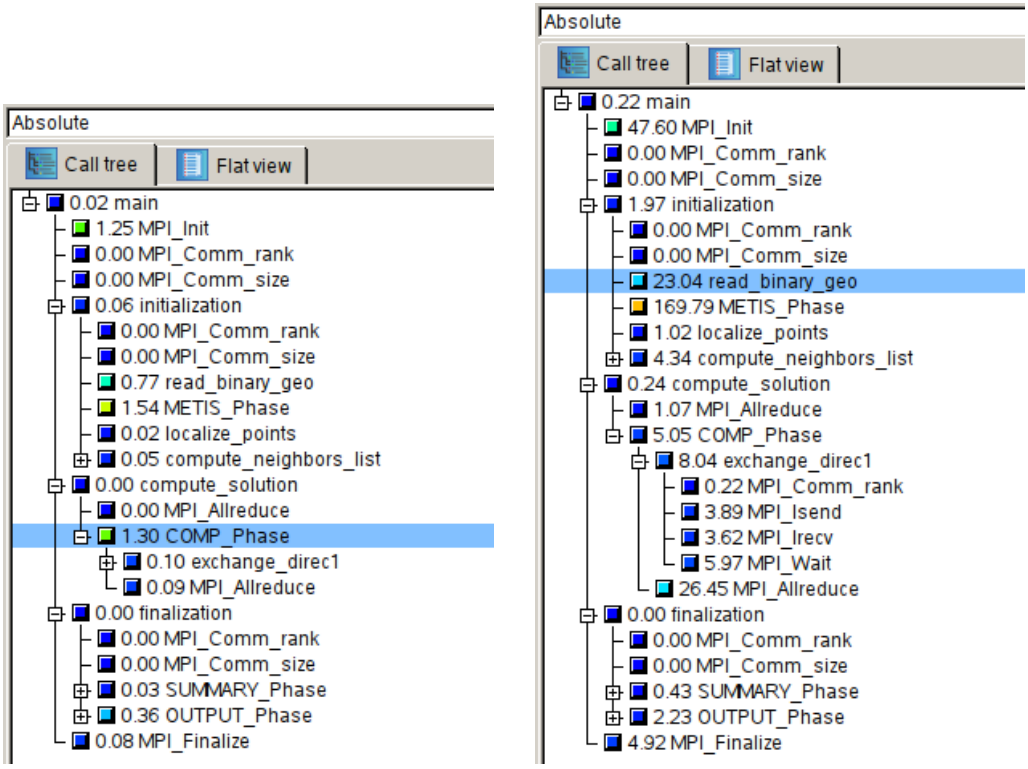
(d) `cojack` data, classical dist.

Figure 4: Speedup graphs.

processes, the overall running time becomes dominated by the initialization phase.

To gain further insight into the causes of performance characteristics, we instrumented the benchmark program using Score-P, and observed the gathered data using *Cube* and *Vampir*.

With Cube, we can see in Figure 5a that communication overhead to the computation phase is $(0.10 + 0.09)/(1.30 + 0.10 + 0.09) = 12.75\%$, which is definitely less than the desired 25%. We also picked a harder case to investigate inefficiencies, namely `cojack` data set running on 64 processes, as shown in Figure 5b. Here METIS dual graph partitioning is the major time consumer overall, and communication overhead of the computation phase is 90.51%. The `MPI_Allreduce` operations take slightly more time than the one-to-one exchange of `direc1` values. Since about 37% of the one-to-one communication time is not spent in MPI calls, we might achieve performance improvements here by using `MPI_Type_indexed` instead of assembling buffers by manual copying. Using Vampir's visualization we can differentiate between collective and one-to-one operations (see Figure 6). Figure 6a suggests that there are consecutive `MPI_Allreduce` calls. We should investigate the source code whether some of them can be merged to a single call.



(a) `pent` running on 4 processes. (b) `cojack` running on 64 processes.

Figure 5: Cube results. Numbers represent the absolute time in seconds.

4.2 Tuning and evaluation

We tried two things to optimize application performance. First, we merged `MPI_Allreduce` calls where it was possible. This optimization was beneficial in case of many processes. With 96 parallel processes the execution time of the computation phase has been reduced by 15% on the `cojack` data set (see Figure 7). Second, we tried to use `MPI_Type_indexed` instead of manually assembling and disassembling buffers, but this approach turned out to be harmful for the performance.

5 Overview

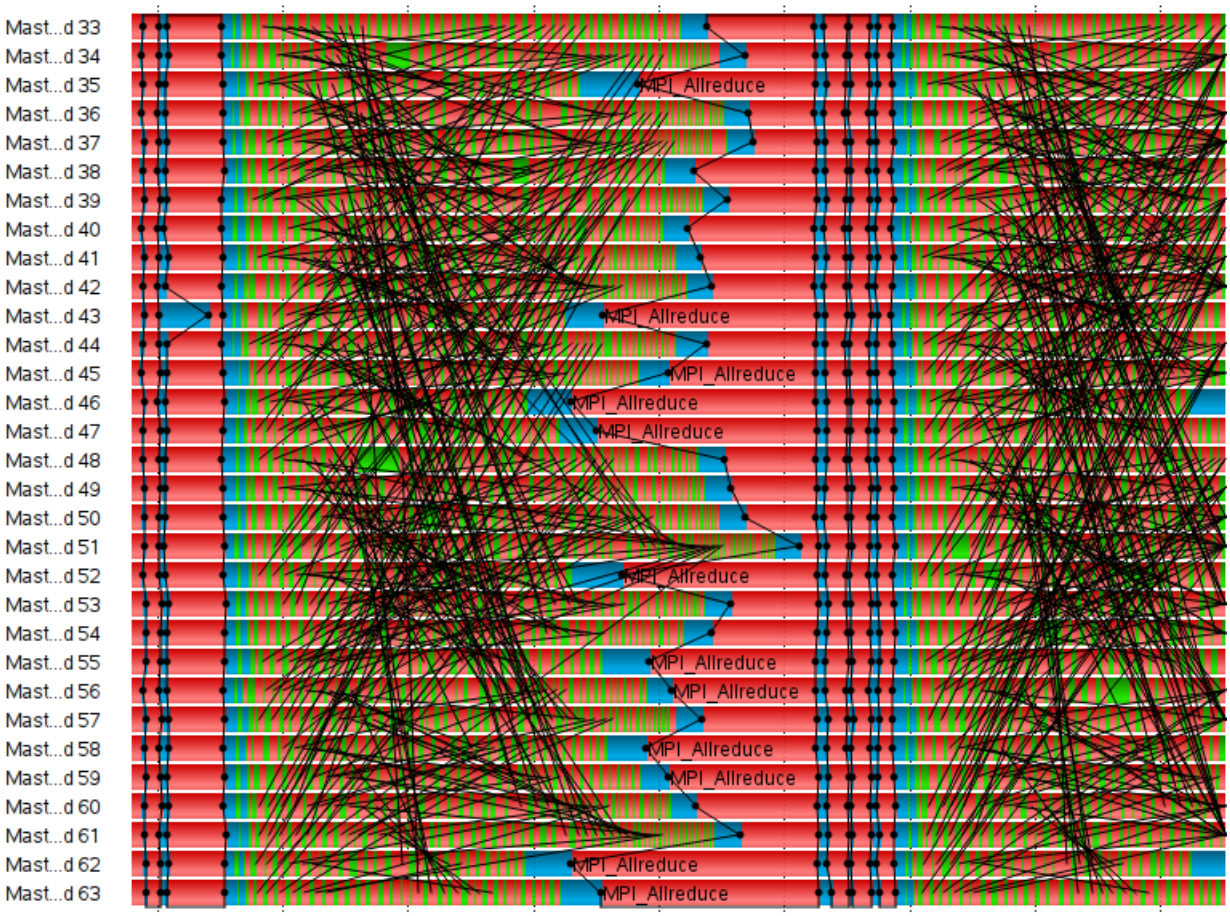
We conclude that the parallelization of the computational phase is fairly good, since in case of strong scaling speedup results will eventually always deteriorate, if there are too many processes. Although there could be some space for improvement, optimization of the I/O phases can have much more effect on the overall performance of the benchmark program.

We also have good scaling for the finalization phase, since we apply distributed output, i.e. every process produces a separate `.vtk` file independently from its own data. We must

note here, that the `.vtk` files are textual, which leaves room for further serial optimization with a binary format. There is also another part of the finalization phase (see Figure 5), namely printing a summary, which gathers all values of the `var` array in one process. This is used for debugging and checking the correctness of the parallelization, therefore in the final product this part could be skipped.

The initialization phase seems to be the toughest to optimize. One obvious try would be to use ParMETIS instead of the serial METIS. If we really want the initialization to scale for large inputs and many processes, then we must use distributed input as well, so that input can be processed independently. However, for a problem, like the generalized orthomin solver, we need not have a very fine description of the geometry just because we want the solution to have fine resolution. We could use a small geometry file, distribute it à la METIS, and then refine the resolution in each process, before the iterative solution.

(a) `pent` running on 4 processes.



(b) `cojack` running on 64 processes.

Figure 6: Vampir results.

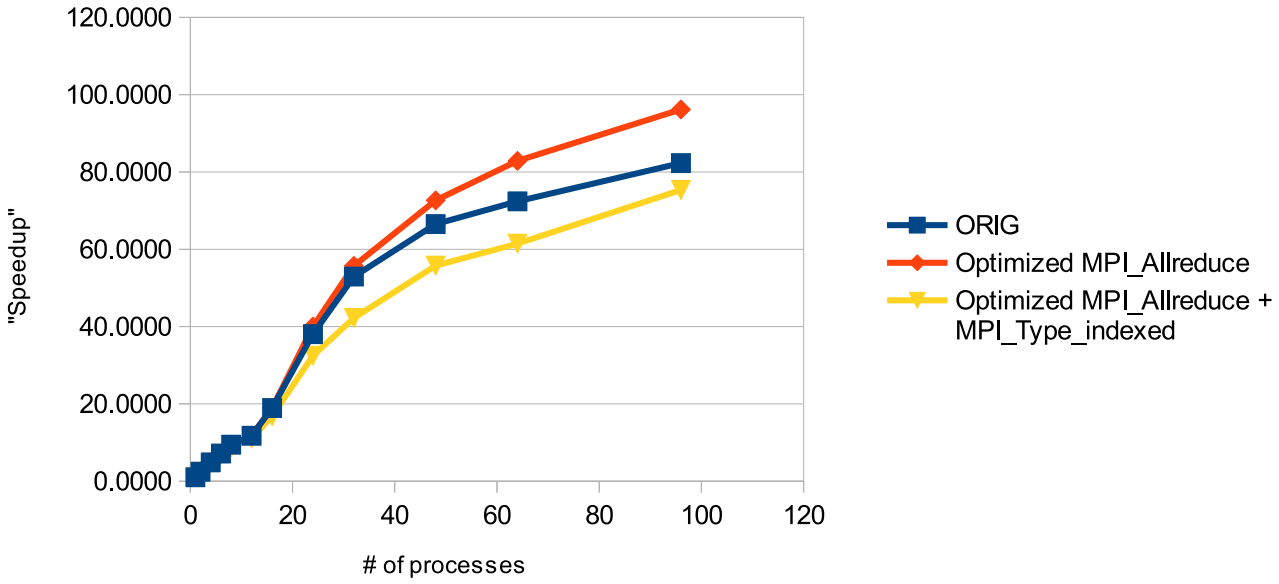


Figure 7: Results of performance tuning for the computational phase. Speedup values are expressed w.r.t. the original version running on 1 process.