

Initial profiling and optimization of single-bunch performance in Synergia 2.1

James F. Amundson
Fermilab, Batavia, IL 60510

Draft 05/26/2011

Abstract

I describe simple profiling and preliminary optimization results for a Synergia 2.1 space charge calculation. My results include a study of MPI collective performance in OpenMPI and MVAPICH2. By optimizing some of the local code and implementing and utilizing a dynamics routine to determine the optimal collective routines at runtime, I have improved peak performance of Synergia 2.1 by a factor of 1.7 for a benchmark problem. I intend these optimizations as a starting point for future, more involved efforts.

1 Introduction

Since Synergia 2.1 contains all-new performance-critical code, profiling and basic performance optimization are important steps to be made before the code reaches production. I describe here a set of profiling exercises of a space-charge calculation using Synergia 2.1 on a Linux cluster. As a result of the profiling, I was able to perform some optimization steps on the largest time-consuming portions of the code. As a result, I have improved the peak performance by a factor of ~ 1.7 . See Figure 1.

2 The benchmark

I used the `cxx_example` benchmark for all results in this document. The benchmark uses the Hockney solver and allows for variable problem sizes. The nominal size for this exercise is a $64 \times 64 \times 512$ space charge grid with 10 particles per cell for a total of 20,971,520 particles. There are 32 evenly-spaced space charge kicks. The single-particle dynamics use second-order maps.

I performed all benchmarks on the intel12 nodes of the Fermilab Wilson Cluster (<http://tev.fnal.gov/index.shtml>). The cluster consists of Intel dual-socket six-core (total 12 cores) Xeon 64-bit X5650 "Westmere" 2.67 GHz nodes connected by single data rate (10 Gbps) Infiniband interfaces. The cluster runs Scientific Linux 5.5. I used the native GCC (version 4.1.2) with the basic `-O3` optimization flag.

3 Profiling

Synergia 2.1 contains some simple profiling code available as a compile-time option. The code can be turned on with, e.g.,

```
cmake -DUSE_SIMPLE_TIMER=true .
```

The profiling code uses `MPI_Wtime` to calculate the execution time of various portions of code on the processor with rank 0. The benchmark program also calculates the total time for the propagate command, whether or not profiling is enabled. With profiling enabled, the propagate time can be used as cross-check to ensure that no significant portions of code are not being profiled.

The simple profiling is as non-intrusive as possible. One limitation of this approach is the possibility that the time reported on processor 0 for collective communications significantly underestimates the time

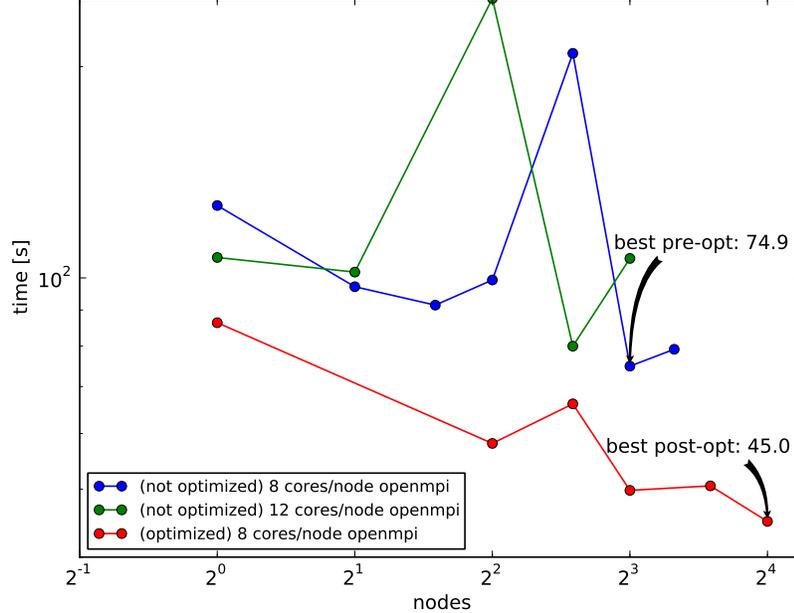


Figure 1: Executive summary: Synergia 2.1 performance before and after the optimization steps described in this document.

for the entire cluster to complete a collective operation. The end result would be an over-reporting of the *next* operation that includes a simple collective. I believe this effect is occurring in some of the later benchmarks in this paper. A more accurate approach will be necessary in subsequent work.

Figure 2 shows the profiled breakdown of the 8 core/node OpenMPI case. Only the routines that account for ≥ 3 percent of the total on the number of cores corresponding to the best performance (64, in this case) are shown. The sum of the remainder are accounted for in the “other” category.

The labels have the following meanings:

sc-get-local-rho The charge deposition step (local).

sc-get-global-rho The charge density communication step (global).

sc-get-phi2 The field calculation, including two forward FFTs, a convolution and one inverse FFT (global). The communication in the parallel FFT is included in this step.

sc-get-global-en The field communication step, in reality the sum of three steps for the three field components (global).

sc-apply-kick The field application step, in reality the sum of three steps for the three field components (local).

Based on these results, I decided to focus on the most important routine for small numbers of cores, kick application (sc-apply-kick in the figure), and the most important obstacles to scaling, charge collection and field distribution (sc-get-global-rho and sc-get-global-en, respectively.)

4 Optimizing kick application

In analyzing the kick applications routines, I found that each kick required multiple function calls in addition to some unnecessary repeated extraction of data. Furthermore, the kick routines require sampling the

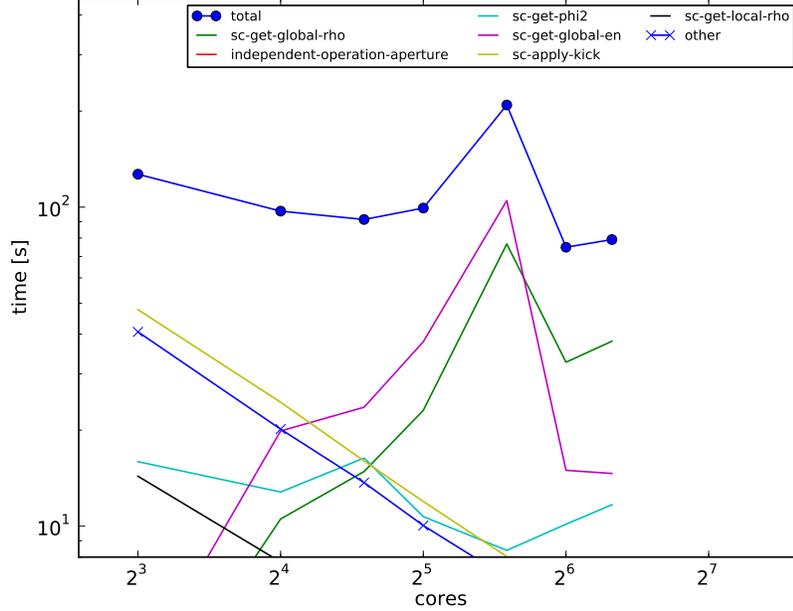


Figure 2: Initial code profile showing routines accounting for at least three percent of total time in the fastest case.

large field data array at different locations for each particle, which must result in cache misses. Finally, the grid application step requires repeated calls to the C function floor, which is known to be slow. I took the following optimization steps:

1. Minimize the number of data extractions, *e.g.*, retrieval of the grid shape.
2. Minimize the number of function calls.
3. Inline all the remaining functions.
4. Add a periodic sort of the particles along the z-axis, in order to make accesses to the field data array more localized. The sorting routine utilizes `std::sort`.¹
5. Add a faster version of floor (`fast_int_floor`.)

These five steps combine to produce an overall speedup of the kick application routines by a factor of ~ 1.9 , as shown in Figure 3.

5 Optimizing communication steps

The charge collection and field distribution communication steps are the biggest barrier to scalability. In order to study the optimal way to perform these steps I considered different combinations of MPI collectives for each routine. I also compared the performance of OpenMPI 1.4.3rc2 *vs.* MVAPICH2 1.6. Finally, I considered the performance differences between using all 12 cores on each node *vs.* using only 8 cores per node.

For the charge collection step, I considered two methods:

¹Thanks to J. Kowalkowski for supplying the necessary adapter code to allow for a simple call to `std::sort`.

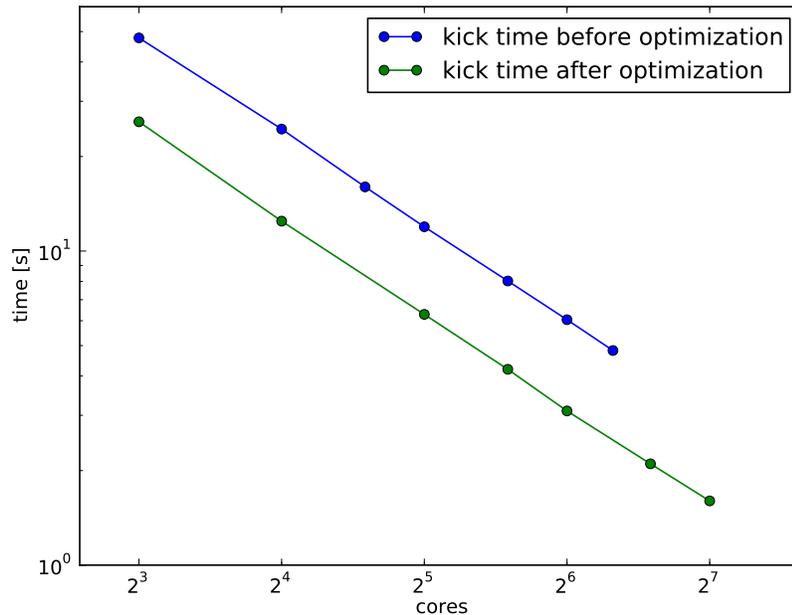


Figure 3: Kick application time before and after optimization

Reduce scatter The method used in the original version of the code. The MPI command `MPI_Reduce_scatter` is used to place the relevant portion of the charge density on the nodes where needed for the parallel field calculation.

Allreduce The `MPI_Allreduce` command is used, which results in much redundant information being made available to each processor.

For the electric field distribution, I considered three methods:

Gatherv Bcast The method used by the original version of the code. The MPI command `MPI_Gatherv` collects the electric field on one processor. The MPI command `MPI_Bcast` then broadcasts the entire vector to all processors.

Allgatherv The MPI command `MPI_Allgatherv` is used to assemble the field on all processors.

Allreduce A field variable is created with all zeros, then the calculated portion on each processor is inserted. The `MPI_Allreduce` command is used to sum the result which is then correct on each processor.

The relative speed of the various methods has a non-trivial dependence on number of cores and MPI implementation. Note that the timings in this section were surrounded by calls to `MPI_Barrier`, so they do not suffer from the simple timer limitations discussed in Section 3. Figures 4, 5, 6, and 7 show the timing results for various combinations of variables.

No single communication routine wins in all cases. I decided to leave all five communication patterns in the code as a runtime option. I also added the method `auto_tune_comm` to test the available communication routines at runtime and select the best option for the given conditions. I used `auto_tune_comm` for the final results.

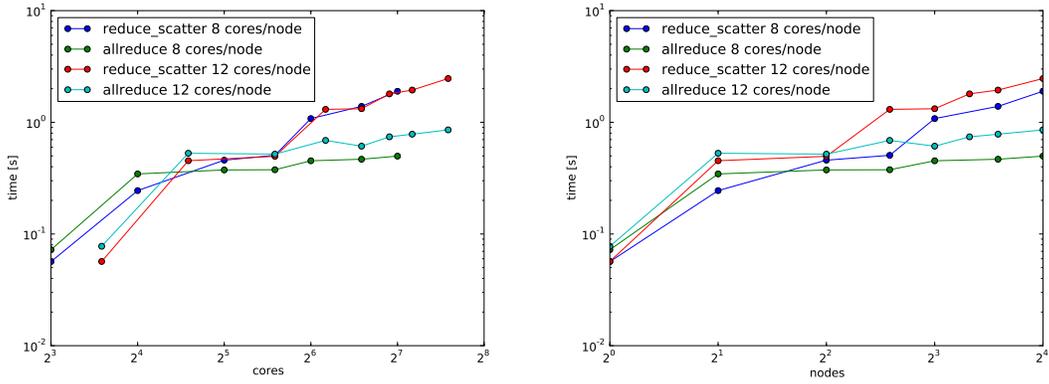


Figure 4: OpenMPI charge collection performance as a function of number of cores (left) and nodes (right).

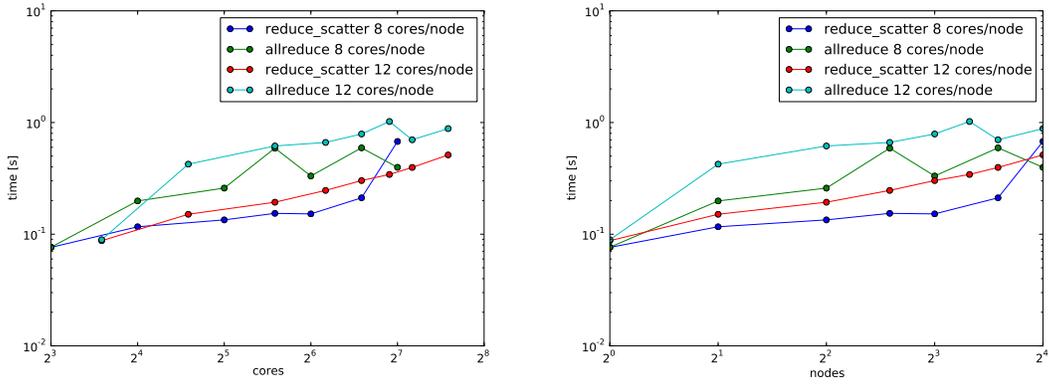


Figure 5: MVAPICH2 charge collection performance as a function of number of cores (left) and nodes (right).

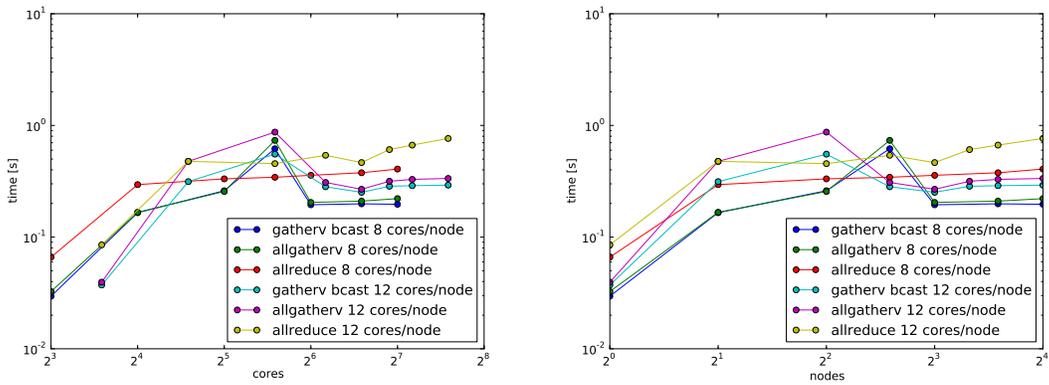


Figure 6: OpenMPI field distribution performance as a function of number of cores (left) and nodes (right).

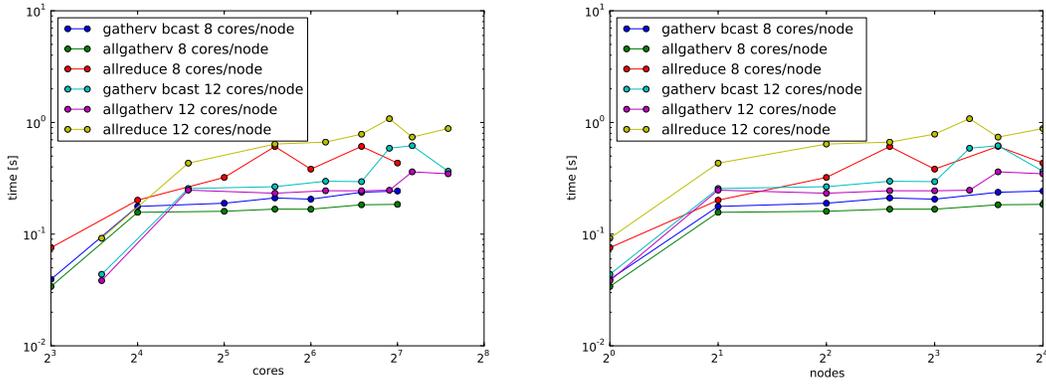


Figure 7: MVAPICH2 field distribution performance as a function of number of cores (left) and nodes (right).

6 Final results for the medium-size problem

Figures 8, 9, and 10 show the results of the optimization steps described above for the medium-size problem.

7 Optimized results for a large problem

The large problem is defined as a $128 \times 128 \times 1024$ grid with 10 particles per cell for a total of 167,772,160 particles.

8 Optimized results for a small problem

The small problem is defined as a $32 \times 32 \times 256$ grid with 10 particles per cell for a total of 2,621,440 particles.

9 Optimized results for a small problem with many particles

The small problem with many particles is defined as a $32 \times 32 \times 256$ grid with 10 particles per cell for a total of 26,214,400 particles.

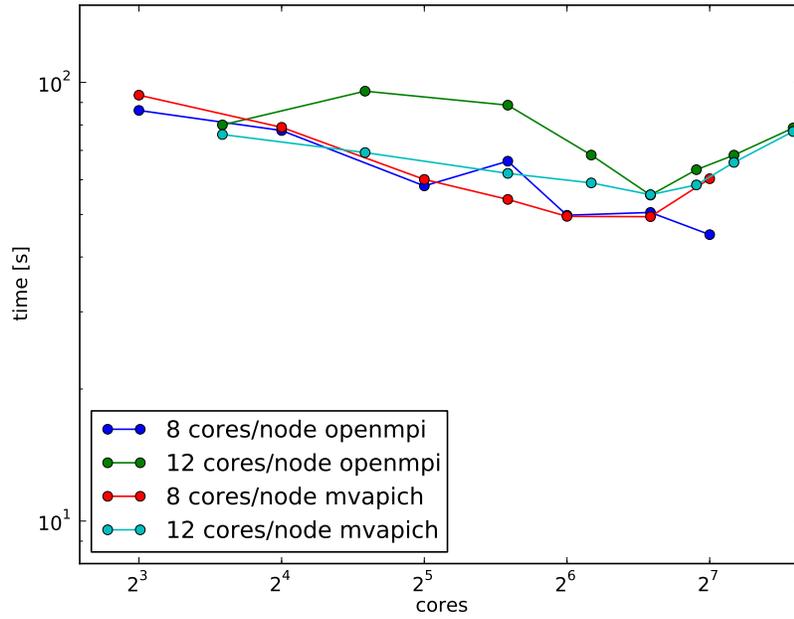


Figure 8: Final result for the medium-size problem as a function of number of cores.

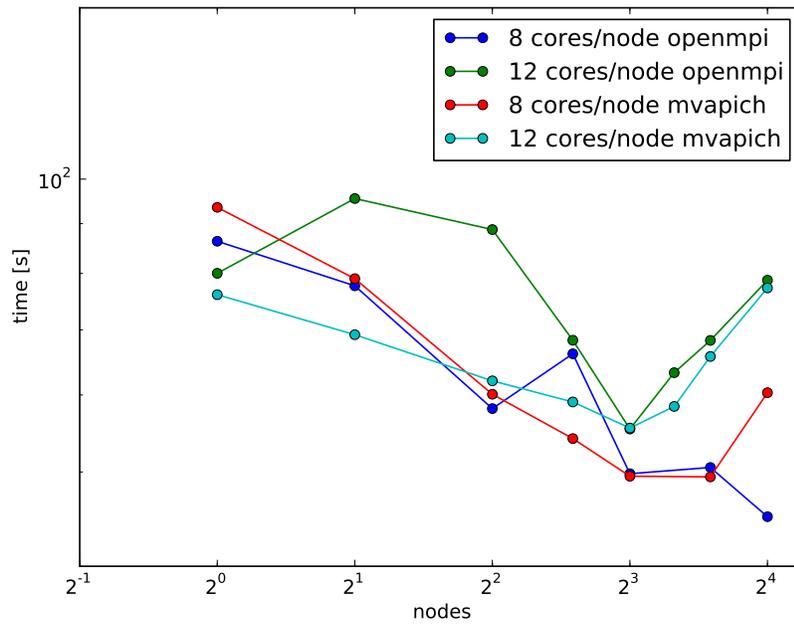


Figure 9: Final result for the medium-size problem as a function of number of nodes.

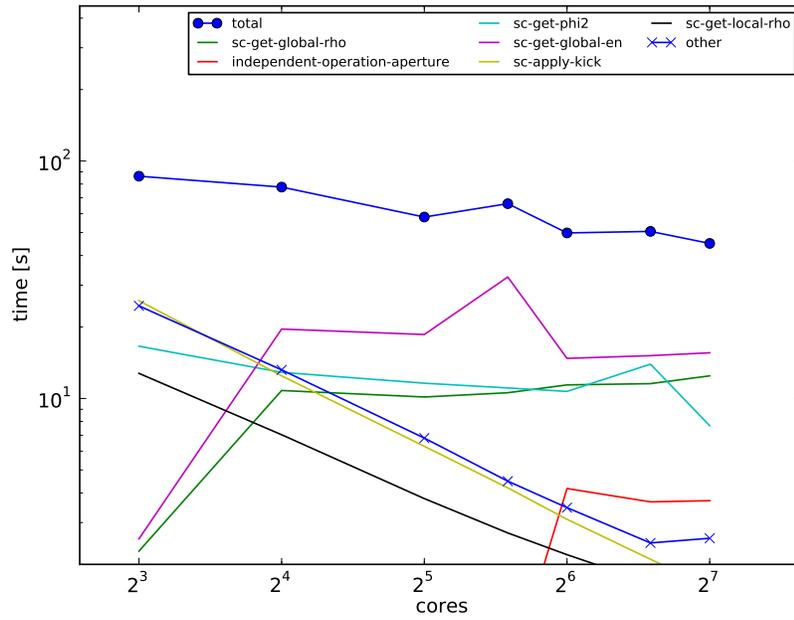


Figure 10: Final (optimized) code profile showing routines accounting for at least three percent of total time in the fastest case.

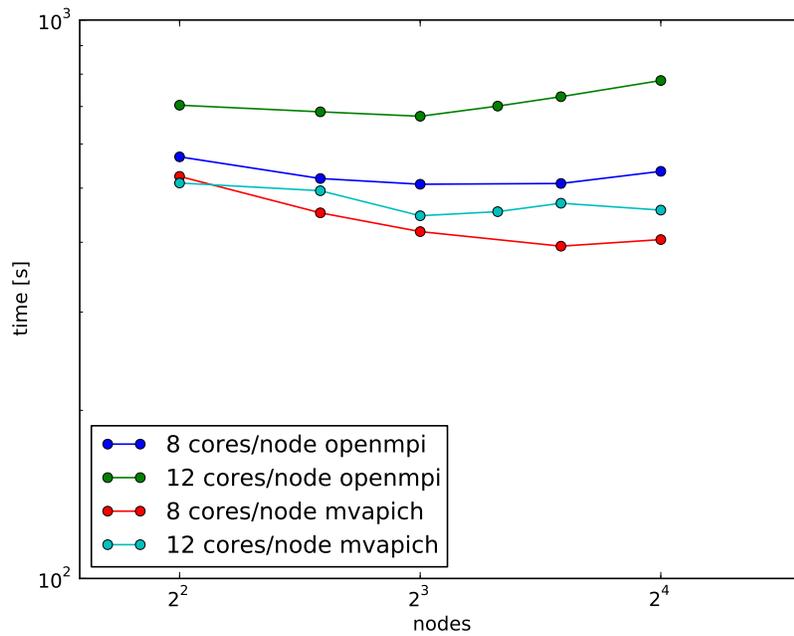


Figure 11: Optimized results for the large problem as a function of number of nodes.

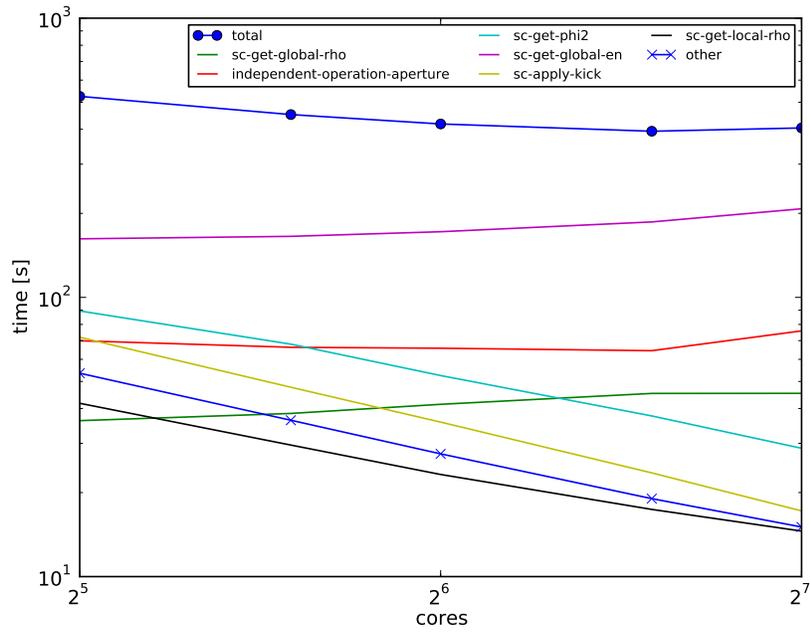


Figure 12: Profiling breakdown for the 8 cores/nodes MVAPICH2 case

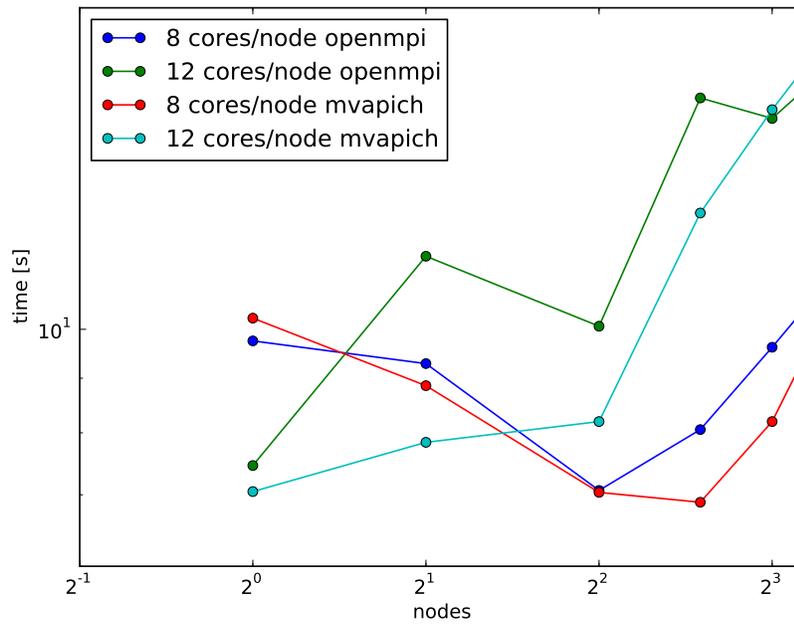


Figure 13: Optimized results for the small problem as a function of number of nodes.

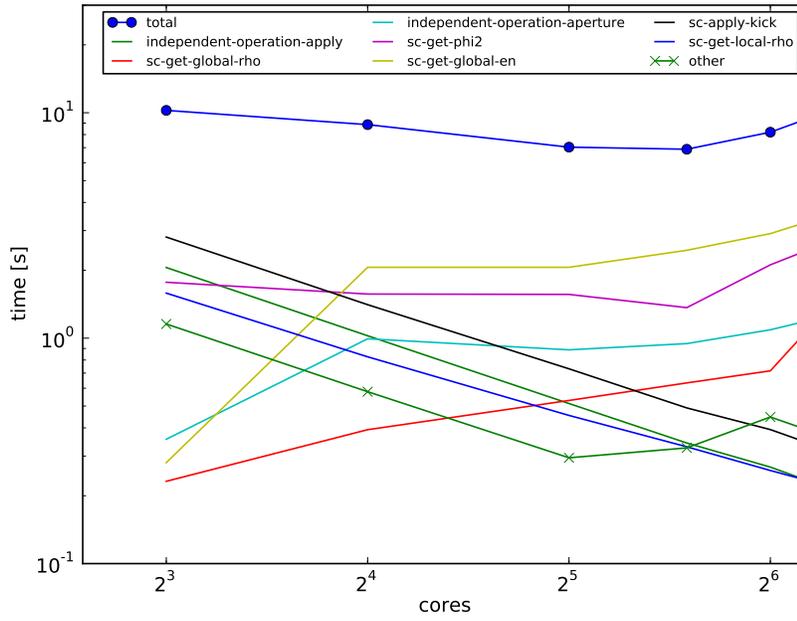


Figure 14: Profiling breakdown for the 8 cores/nodes MVAPICH2 case

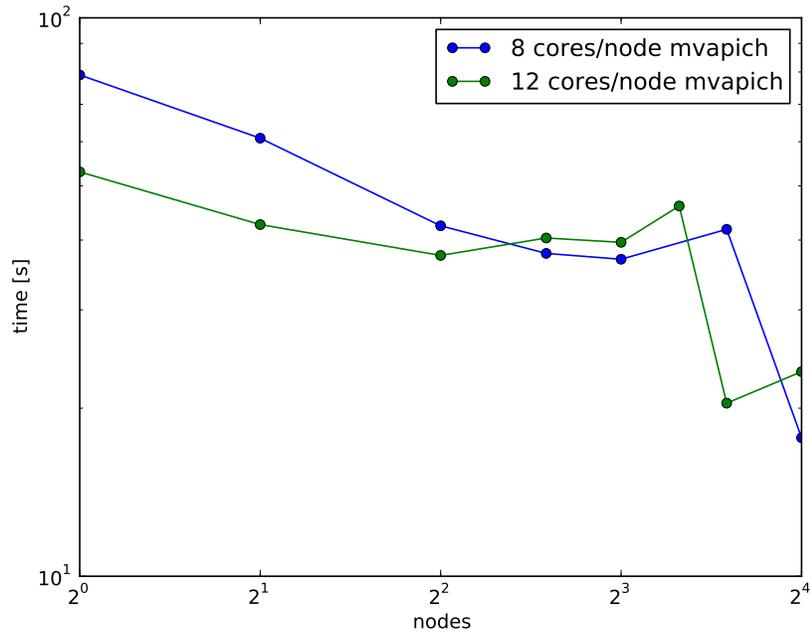


Figure 15: Optimized results for the small problem as a function of number of nodes.

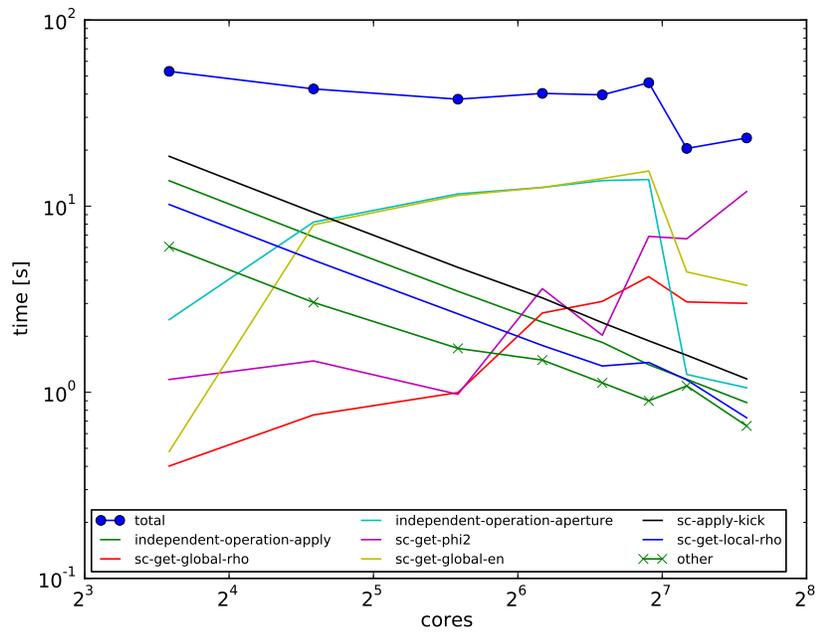


Figure 16: Profiling breakdown for the 8 cores/nodes MVAPICH2 case