MPQC: Performance Analysis and Optimization

Abhinav Sarje, Samuel Williams and David H. Bailey
Computational Research Division
Lawrence Berkeley National Laboratory, Berkeley, CA
{*asarje,swilliams,dhb*}*@lbl.gov*

January 24, 2013

Abstract

MPQC (*Massively Parallel Quantum Chemistry*) is a widely used computational quantum chemistry code. It is capable of performing a number of computations commonly occurring in quantum chemistry. In order to achieve better performance of MPQC, in this report we present a detailed performance analysis of this code. We then perform loop and memory access optimizations, and measure performance improvements by comparing the performance of the optimized code with that of the original MPQC code. We observe that the optimized MPQC code achieves a significant improvement in the performance through a better utilization of vector processing and memory hierarchies.

1 Introduction

MPQC [1], which stands for *Massively Parallel Quantum Chemistry*, is a widely used computational quantum chemistry [2] code to compute electronic structures and properties of atoms and molecules. It provides a number of *ab-initio* methods to do so including Hartree-Fock method (HF) and MøllerPlesset perturbation theory (MP) [3], based on the Schrödinger equation.

The Hartree-Fock method is central to computational quantum chemistry, and one of the simplest methods. It is primarily used in electronic structure calculations. It performs calculations based on the averaged effect of electron repulsions, or mean field, and does not take into account the exact electron repulsion effects. The results of these calculations are approximate atomic energies expressed in terms of the system's wave function. Because of the approximate calculations, HF is one of the fastest methods for these computations. HF calculations form the foundation for many other sophisticated electronic structure computations. One such method is the Møller-Plesset perturbation theory, which corrects results of HF through computation of electron-electron repulsion energies, or electronic correlation. MP is based on Rayleigh-Schrödinger perturbation theory, and usually defined to the second order (MP2), third order (MP3), or fourth order (MP4). It was published way back in 1934 [4]. The most time consuming, but one of the most important part in such computations (HF, MP2) is evaluation of two-electron integrals [5] [6] [7], also known as electron repulsion integrals. The electronic structure computations are performed using a *basis set* [6]. A basis set is a set of basis functions which are used to create molecular orbitals. These basis functions are generally atomic orbitals which are taken in a linear combination to form molecular orbitals. A minimal basis set defines only basic aspects of the orbitals. Slater Type Orbital (STO) basis functions form minimal basis sets. Further, an extended basis set considers higher orbitals of molecules and accounts for their sizes and shapes. Examples of extended basis sets are Double-Zeta ($\texttt{cc-pVDZ}$), Triple-Zeta ($\texttt{cc-pVTZ}$), Quadruple-Zeta ($\texttt{cc-pVQZ}$), Quintuple-Zeta ($\texttt{cc-pV5Z}$) and Sextuple-Zeta ($\texttt{cc-pV6Z}$) basis sets.

MPQC provides an implementation for calculation of Hartree-Fock energies and gradients for various conditions, and second order Møller-Plesset perturbation theory, including among few other methods. MPQC is written mostly in C++ and is based on the object-oriented model. MPQC's input format is as well based on object-oriented model. It also provides support for parallel computations through both message-passing and shared-memory models.

In this report, we present a detailed performance profiling and analysis of MPQC code, mainly for computations of HF and MP2 energies. We also present optimization of one of the functions picked through the performance data obtained.

2 MPQC Inputs

The primary input to MPQC is an object oriented format file which defines all the input objects to be fed to the computations. For information on the input format, and further details on the inputs to MPQC, refer to the MPQC documentation. For our purposes of performance analysis of MPQC, we will only focus on two components of the input: the molecule for which energies are to be calculated, and a basis set which we take from the class of Gaussian basis sets. Integral evaluation time complexity increases steeply with the size of the input molecule. For example, computation of MP2 energies grows as $O(n^8)$, where n is the molecule size. In this report we consider the following input molecules:

	Name	Formula	Mol. Mass ≈
1.	Water	H_2O	18
2.	Water dimer	H_4O_2	36
3.	Methane	CH_4	16
4.	Ethane	C_2H_6	30
5.	Butane	C_4H_{10}	58
6.	Octane	C_8H_{18}	114

We use some of the commonly used Gaussian basis sets in our experiments presented in this report. One is an STO basis set, and others are Quadruple-Zeta, Quintuple-Zeta, and Sextuple-Zeta. These are listed below along with the total number of basis functions in each set:

	Basis Set	Num. Basis Functions
1.	`STO-3G`	15
2.	`cc-pVQZ`	144
3.	`cc-pV5Z`	241
4.	`cc-pV6Z`	375

Further information on the these basis sets can be found along with the basis sets distributed along with MPQC in the MPQC documentation for the GaussianBasisSet class [8]. We use these molecules and basis sets inputs to calculate Hartree-Fock energies, and second order MP (MP2) energies of the respective molecules.

3 Experimental Environments

We use two main environments to execute MPQC in our experiments. The first is a IBM iDataPlex system linux cluster. Each node in this cluster contains two quad-core Intel Xeon X5550 processors (based on the *Nehalem* microarchitecture), hence providing a total of 8 cores on each node. The clock speed is 2.67 GHz, and each node is equipped with a total of 24 GB 1333 MHz DDR3 memory. Each processor has a 8 MB of L3 cache shared among all 4 cores, and 256 KB of L2 and 64 KB (32 KB instruction + 32 KB data) of L1 caches per core. Peak theoretical memory bandwidth is 32 GB/s. The nodes in this cluster are connected by 4X QDR InfiniBand interconnects. We run our various scaling performance analysis experiments, which we will describe later, on this platform.

The second platform we use is a single node linux system, with two hex-core 2.93 GHz Intel Xeon X5670 processors (based on the *Nehalem* microarchitecture, codenamed *Westmere*), providing a total of 12 cores. This system is equipped with 94 Gb of DDR3 main memory. Each on-board processor has 12 MB of L3 cache shared among all 6 cores, and 256 KB sized L2 and 64 KB sized (32 KB instruction + 32 KB data) L1 caches per core. The maximum memory bandwidth is 32 GB/s. We mainly utilize this system for low level profiling and optimization analysis experiments.

4 Function-level Performance Analysis

We will start a detailed performance profiling and analysis of MPQC with details on the execution times of various functions in the program, their scaling with respect to the number of nodes and threads, and the input basis sets. To do so, we use a number of profiling tools available, but we will only present the results obtained through the profiling tool TAU [9]. In order to obtain data

using TAU, the MPQC code had to be compiled by TAU compiler wrappers with TAU libraries, and executed using the TAU runtime framework.

Recall that MPQC is capable of both message-passing and threading parallelism, where generally one would want to have message-passing parallelism across a number of compute nodes, and threading parallelism within a node across all the cores. MPQC implements message-passing using MPI, and threading using pthreads. In this section we use the first test environment – the linux cluster. On this platform we can take advantage of both, a number of compute nodes, and at most 8 cores per node. Given this architecture and the capabilities of MPQC, we are faced with four main parallel configurations for the execution of MPQC. These configurations are described below.

1. We call the first configuration *"single-single"* (SS), where a single MPI process is executed on each compute node, and each MPI process using a single thread. Hence, only one core of each node is utilized in this configuration.

2. In the second configuration, *"multiple-single"* (MS), each compute node executes multiple MPI processes, and each MPI process using a single thread. In this configuration, we execute one MPI process per node (each with a single thread), making total of at most 8 MPI processes (for 8 cores) per compute node, utilizing all the cores.

3. The third configuration, *"single-multiple"* (SM), consists of each compute node executing a single MPI process, and each MPI process executing multiple threads. In our case we used 8 threads per MPI process, hence utilizing all the available cores.

4. The fourth configuration is the most general, *"multiple-multiple"* (MM), where each node runs multiple MPI processes, each process executing multiple threads. In our experiments with 2 MPI processes per node and 4 threads per MPI process, we noticed that the results obtained in this configuration were almost the same as in the "single-multiple" configuration, hence we will skip the data from this configuration in the following.

We present the performance profiling and scaling results of MPQC for each of the first three configurations one-by-one in the following. We then compare all these configurations by comparing the presented results. We will first focus only on the execution time based profiling. We use the water molecule (H_2O) as input to compute its Hartree-Fock energies using two different basis sets, $\texttt{cc} - \texttt{pVQZ}$ and $\texttt{cc} - \texttt{pV5Z}$.

4.1 Configuration SS

We begin by presenting the scaling of MPQC computing the Hartree-Fock energies of H_2O. In Fig. 1 we show log-log plots of total execution time scaling with varying number of cores used. The total time spent in MPI-routines and non-communication components of the total execution time are also shown as a function of the number of cores. The distribution of time consumed by various routines across all the nodes is also shown for the case when number of nodes is 8.

Note that the larger the basis set used, the better the program scales. With $\texttt{cc} - \texttt{pVQZ}$, the code scales well until 32 cores (or, compute nodes in this case), are used. But the MPI-communication time surpasses the local time at 16 cores itself. Therefore, we see that the total execution time scales well only on small number of cores, after which it starts to increase due to the increased MPI time. Similarly, with the larger basis set $\texttt{cc} - \texttt{pV5Z}$, this behavior can be seen, but shifted to the right. It is able to scale well until 32 cores, after which on 64 cores, the MPI time becomes more significant than the local time. Also note the large variations in time taken by the MPI time across all the nodes. This imbalance also contributes to the overall MPI time.

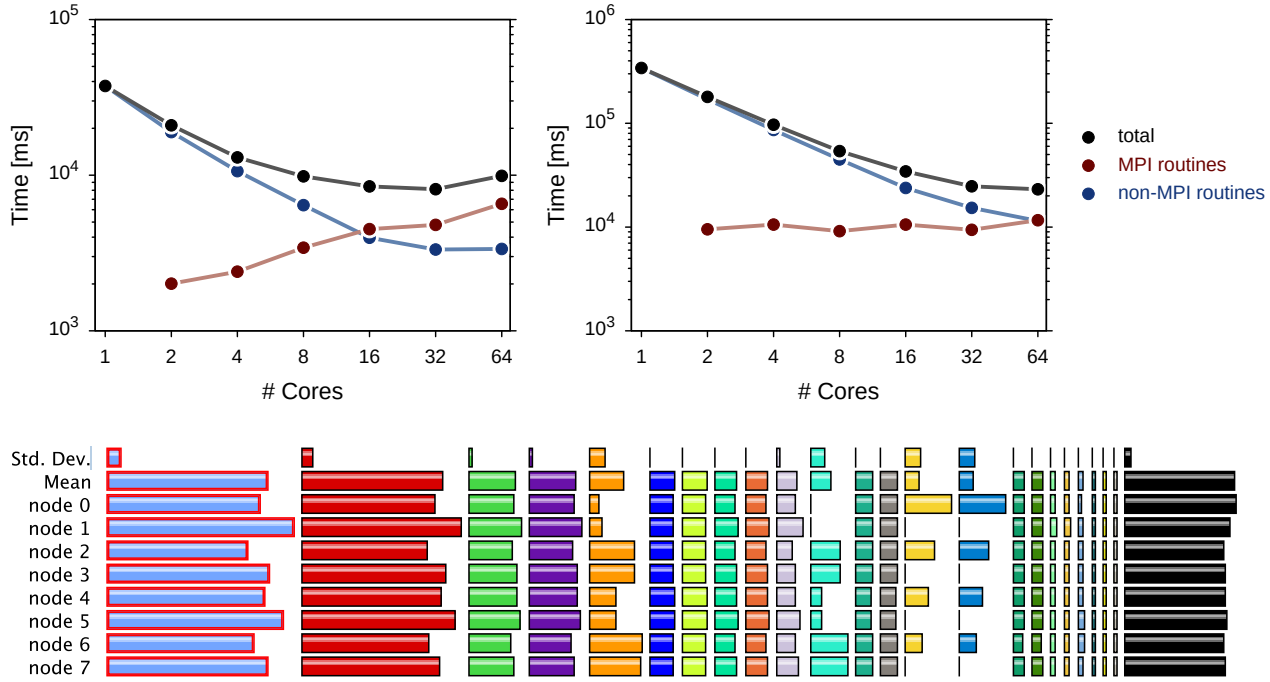

Figure 1: Configuration SS: One MPI process per compute node and one thread per MPI process. The plots on the top show strong scaling of MPQC w.r.t. the number of cores used (which is equal to number of nodes in this configuration), for computing HF energies of H_2O. The plot on the left was obtained when using the basis set `cc-pVQZ`, while the one on the right was using `cc-pV5Z`. Note that 7 cores per compute node are idle. The bar chart on the bottom shows the distribution of execution times across all nodes when 8 nodes are used, along with mean and standard deviation. Each color represents a different routine – light blue and red: `blockbuildprim`(), green and violet: `compute_erep`(), orange: `MPI_Allreduce`(), blue and light green: `init_pure`().

Bad scaling due to MPI-routines in this configuration is also expected since only one core per compute node is used while the remaining 7 cores sit idle. Hence, all the MPI communication goes over the interconnect network. We will now further investigate the scaling by breaking down these timing results into smaller components – routines of MPQC executed during a run.

4.1.1 Non-communication Routines

To get a further insight into the bottlenecks in the program, a breakdown of time spent locally on a compute node is shown in Fig. 2 as log-log plots. Only the top few most time consuming routines are shown. The figure contains scaling of these routines with respect to the number of cores.

It can be clearly seen in this figure that the most time consuming routines which are listed, scale quite well w.r.t. number of cores used. One would note that there is one exception. The time consumed by the routine `init_pure` remains almost the same irrespective of the number of cores.

Due to this, it is clear that at point on the number of cores axis, the time spent in this routine will be more significant than all others, thereby limiting the scaling of the application. This point occurs, in the plots in Fig. 2, around 8 cores in the case of basis set `cc-pVQZ`, and around 32 in the case of `cc-pV5Z`. At this point, the total time spent in non-communication routines will flatten out, and will make this routine a major bottleneck, as can be seen at points with 16, 32 and 64 cores in the first case.

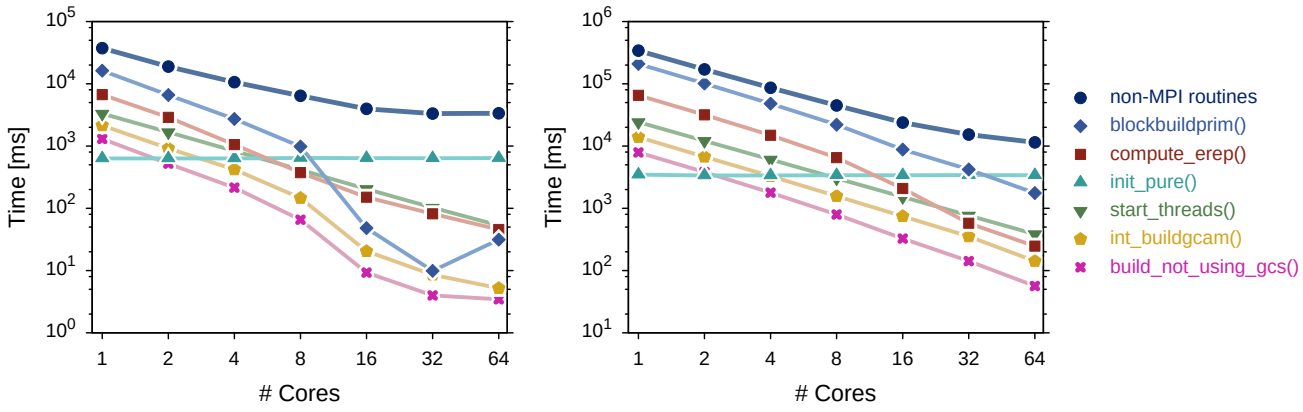

Figure 2: Breakdown of total execution time in non-MPI component into times consumed by various routines in MPQC in configuration SS. On the left is a plot for an execution while using the basis set `cc − pVQZ`, and on the right is while using `cc − pV5Z`.

In both these results it can be seen that the routines `blockbuildprim`() and `compute_erep`() are the two most time consuming routines for most of the cases. It can also be seen that these two routines show super-linear scaling around 16 cores.

4.1.2 MPI Routines

A breakdown of the total time spent in various MPI routines is shown as a log-log plot in Fig 3, in a similar fashion as for non-communication routines above. Scaling of the amount of time taken by a few top time consuming MPI routines is shown in this figure.

It can be noted from Fig. 3 that the total time consumed by MPI routines increases with the number of cores, although in the case of using `cc − pV5Z`, this increase is marginal. This behavior is expected in most MPI programs since the amount of communication increases as the number of nodes is increased. `MPI_Allreduce`() is the most time consuming routine among all MPI routines used in MPQC. It is followed by `MPI_Bcast`() and the difference in times of these two routines decreases with increasing number of cores. Basically, `MPI_Bcast`()'s rate of increase in time is higher than the former. Other routines shown, such as `MPI_Init_thread`() and `MPI_Reduce`() have an increasing rate of increase in time. The time consumed by these with lower number of cores is comparatively insignificant, but becomes quite significant as the number of cores are increased.

It can also be seen that the amount of time consumed by `MPI_Allreduce`() remains nearly the same except for the case when going from 2 to 4 cores with the larger basis set. It can therefore be seen that other MPI routines become more significant in comparison for larger number of cores.

Note that these are strong scaling results. Hence even though the number of messages may increase with increasing number of nodes (increasing total amount of latency for communication setup), the sizes of these messages would decrease (and hence, decrease the message transmission times). This can probably explain the behavior of `MPI_Allreduce`() and `MPI_Bcast`() in flattening out. It will also explain the behavior for the initialization routine in that the number of initializations increase with number of nodes, and, hence, the time consumed by each initialization would also increase.

The above breakdown into MPI routines makes it more clear as to how the overall execution times are affected, and how the times consumed by MPI routines eventually become significant and dominant compared to the non-communication routines. Fig. 5 contains the mean execution times for some of the most time consuming routines.

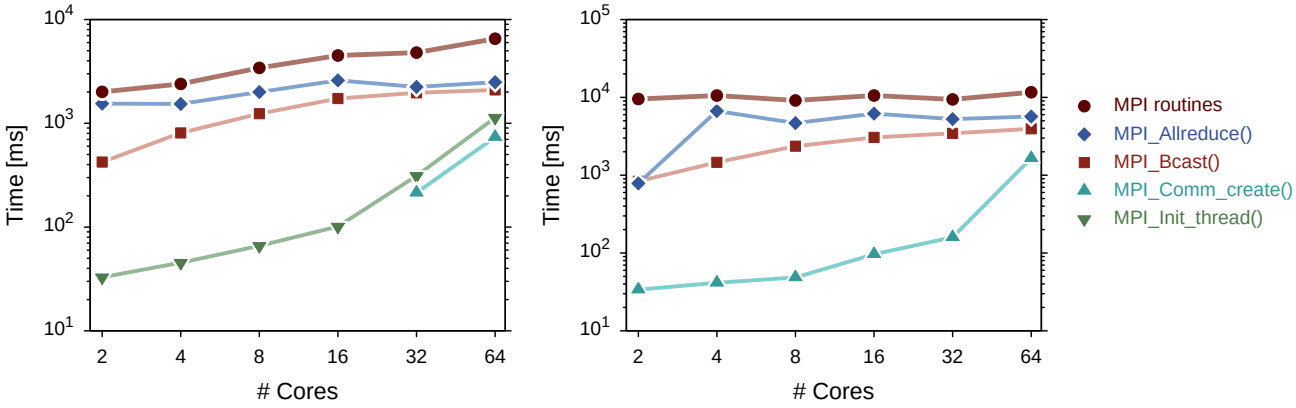

Figure 3: Breakdown of the total execution time spent in MPI routines into times consumed by each of the individual routines in MPQC in configuration SS. Only the top few most time consuming MPI routines are shown which are candidates for bottlenecks in the program. On the left is a plot obtained using the smaller basis set $\texttt{cc-pVQZ}$, and on the right is one obtained using the larger basis set $\texttt{cc-pV5Z}$.

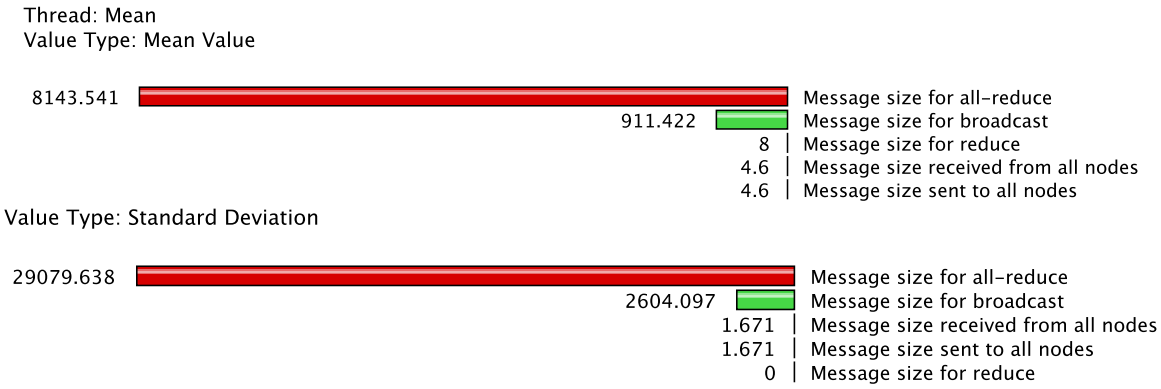

Figure 4: Mean message sizes and their standard deviation for various MPI routines when 8 nodes are used. Note the high variation in the message sizes for all-reduce and broadcast routines.

4.2 Configuration MS

In the second configuration, we execute 8 MPI processes per node, 1 process per core, and a single thread for all. In Fig. 6 we show log-log scaling plots of total execution time scaling with varying number of cores. The total time spent in the two components, MPI routines and non-MPI routines, are also shown as a function of the number of cores. The distribution of the execution times for various routines across the cores used is also included.

We see a very similar behavior here as for the first configuration SS. The program scales better with larger basis set used. In the case of $\texttt{cc-pVQZ}$, the code scales until 8 cores, beyond which the total execution time flattens out and eventually increasing. In the case of the larger basis set $\texttt{cc-pV5Z}$, the code scales well until 32 cores. We observed similar numbers for configuration SS earlier. The two components of the total time also show similar behavior. The crossover of total time taken by MPI routines and non-MPI routines happens at 32 cores for smaller basis set and at 64 for the larger basis set. Therefore, we again see that the total execution time scales well only on small number of cores. Note that in this configuration we obtained data for the same number of compute nodes as

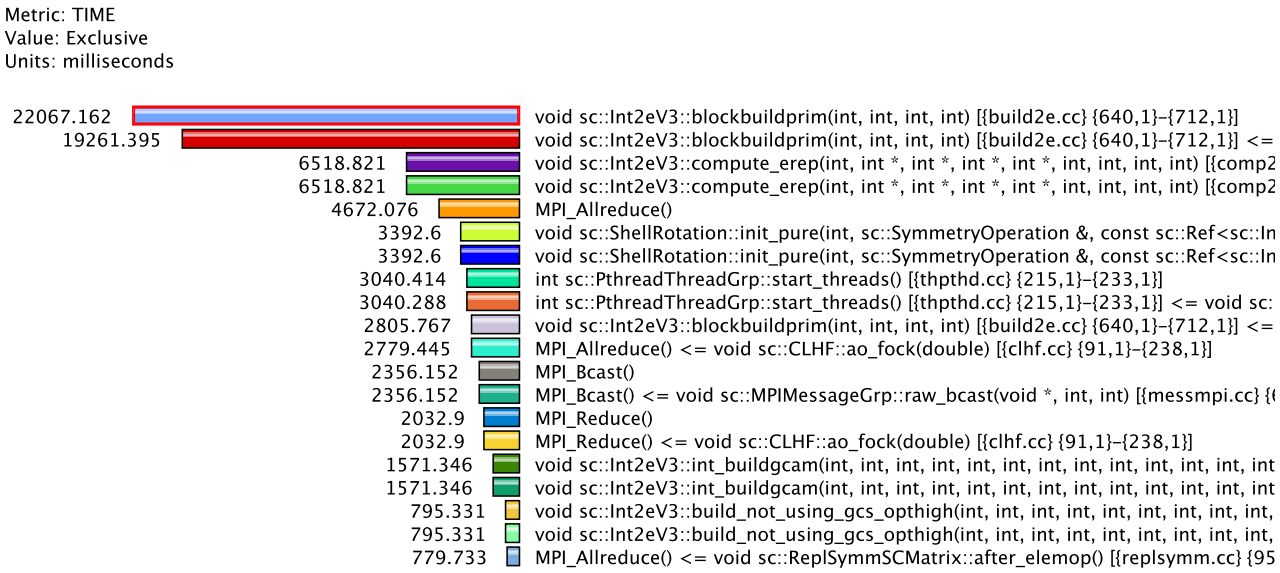

Figure 5: Top few mean values of execution times taken by various MPI and non-MPI routines in configuration SS when 8 nodes are used.

in configuration SS, giving us a total of 512 cores for 64 nodes. Distribution of the times taken by various routines across the cores used also shows a very similar pattern as for configuration SS, with unbalanced MPI communication times.

In the following we will now investigate the scaling further by breaking down these timing results into individual routines.

4.2.1 Non-communication Routines

A breakdown of the amount of time spent locally on a compute node is shown in Fig. 12 as log-log plots, scaling w.r.t. the number of cores used. Only the top few most time consuming routines are considered.

It can be seen that the listed most time consuming routines, scale quite well w.r.t. number of cores used. We again see one exception, routine `init_pure`() consumes almost same amount of time irrespective of the number of cores used. Due to this, the overall scaling for non-MPI routines is limited at 16 cores and 128 cores, respectively, for the smaller `cc-pVQZ` and larger `cc-pV5Z` basis sets. We note that these points occur for larger number of cores in this configuration compared to configuration SS.

Again, in both these results the routines `blockbuildprim`() and `compute_erep`() are the two most time consuming routines for most of the cases, and the former routines has a higher rate of super-linear scaling compared to the latter beyond 8 and 64 cores in the two cases respectively. In this case, some other routines (`int_buildgcam`() and `build_not_using_gcs`()) also show this super-linear scaling behavior.

4.2.2 MPI Routines

A breakdown of the total time spent in few most time consuming MPI routines is shown as a log-log plot in Fig 8 for the two cases of using basis sets `cc-pVQZ` and `cc-pV5Z`.

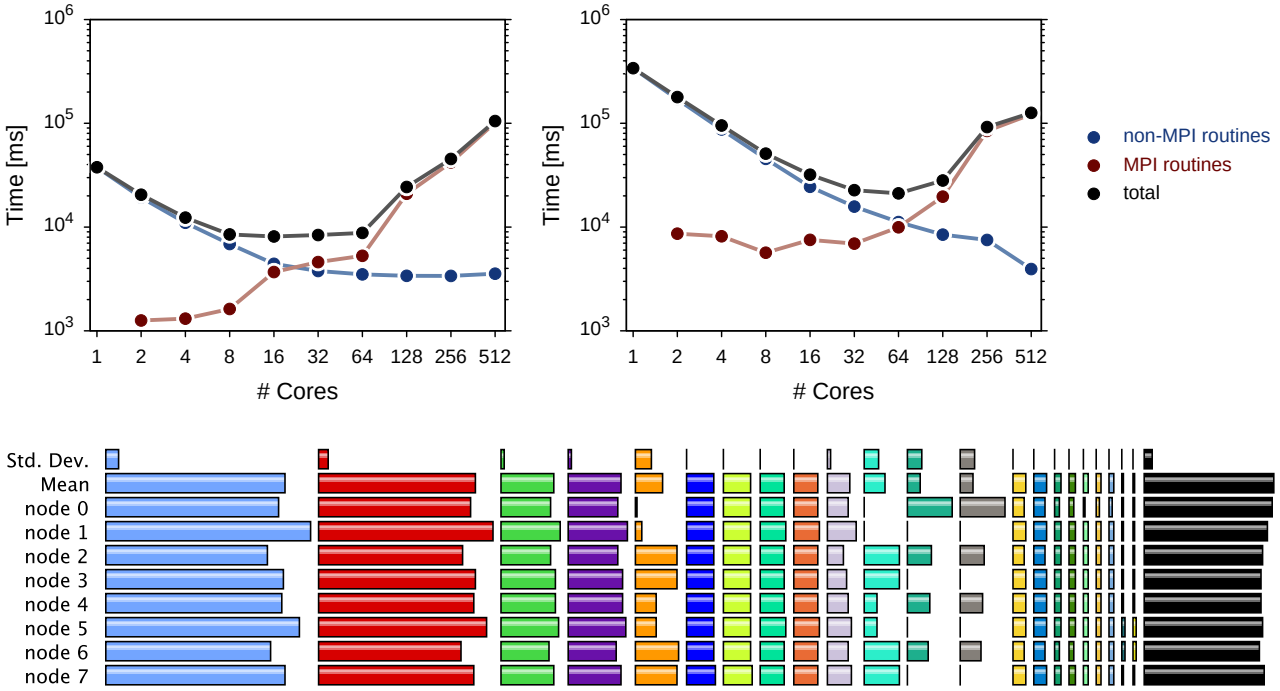

Figure 6: Configuration MS: 8 MPI processes per compute node, one thread per MPI process. Strong scaling w.r.t. the number of cores used for computing HF energies of H_2O is shown in the plots on the top. The plot on the left was obtained when using the basis set `cc-pVQZ`, while the one on the right when using `cc-pV5Z`. In this case, all the cores on a node are utilized. The bar chart on the bottom shows the distribution of execution times across 8 cores.

As seen Fig. 8, the total time consumed by MPI routines increases with the number of cores. This increase is marginal for smaller number of cores used.

Similar to configuration SS, `MPI_Allreduce`() is the most time consuming routine among all MPI routines here, followed by `MPI_Bcast`(). In this configuration we are able to see the points where `MPI_Bcast`() becomes the most dominant MPI routine. This happens at 64 cores in both cases as seen in the figure. We also see a similar behavior for `MPI_Allreduce`(). Fig. 9 shows the mean and standard deviations of the message sizes for various MPI routines. We again see a similar behavior as in for configuration SS. We show the exclusive times taken by top few most time consuming routines in Fig. 10. Here also we see similar pattern as the previous configuration.

4.3 Configuration SM

This configuration incorporates the use of multi-threading. Here, each compute node executes a single MPI process, and each MPI process executes 8 threads. Plots of total execution time, total time spent in MPI routines, and non-MPI routines in this configuration are shown in Figure 11 as functions of the number of cores used. In this case as well all 8 cores on a node are utilized.

In this configuration we see quite different behavior compared to the first two configurations SS and MS. Overall, the code scales until 32 cores for the case with `cc-pVQZ` basis set, and 64 for the case with `cc-pV5Z` basis set, after which the execution time flattens out. In this case we see that non-MPI routines do not scale well, as was the case earlier, beyond certain points. Both non-MPI routines and MPI routines show a slow down as the number of cores is increased beyond these points. We also note that the time consumed by MPI routines always remains insignificant compared to the times

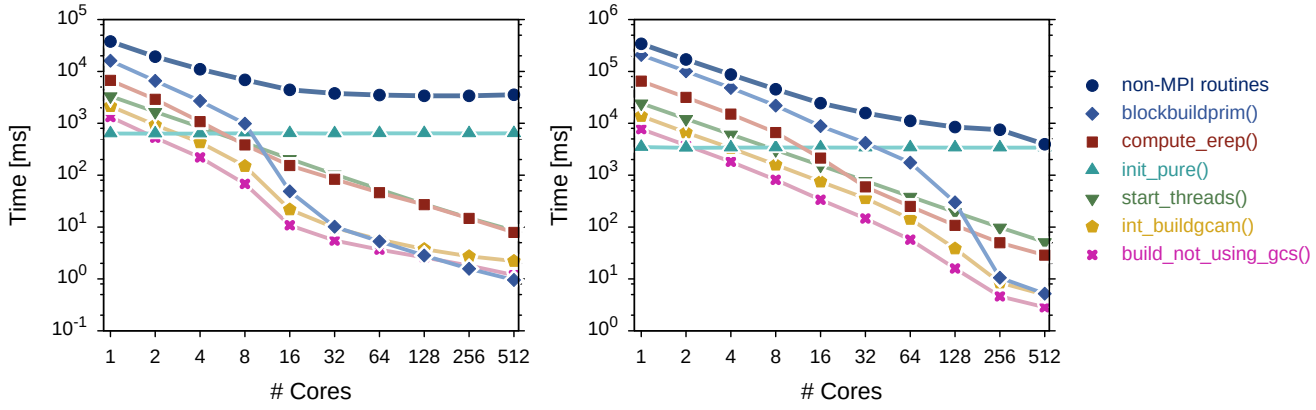

Figure 7: Breakdown of the time spent in non-MPI routines into times consumed by each individual routine in configuration MS. On the left is a plot for an execution using the basis set `cc-pVQZ`, and on the right is one using `cc-pV5Z` basis set.

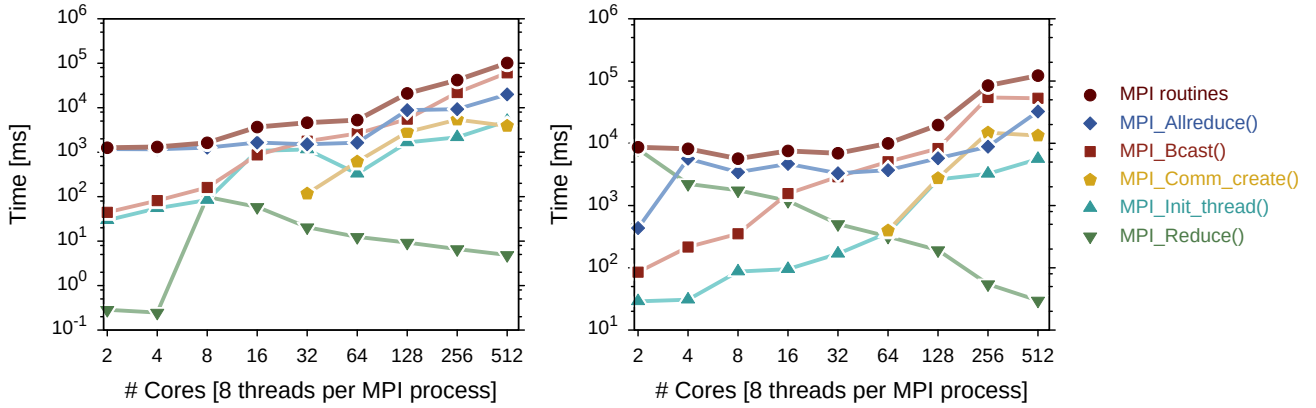

Figure 8: Breakdown of the total execution time spent in MPI routines into times consumed by each of the individual routines in configuration MS. On the left is a plot obtained using the smaller basis set `cc-pVQZ`, and on the right is one obtained using the larger basis set `cc-pV5Z`.

consumed by non-MPI routine, and we never see the crossovers we saw in previous configurations. The shape of the total execution time curve is mostly defined by the non-MPI routines which also show a similar curve shape.

Let us now break down these timing results into individual routines in order to investigate the reasons behind such a different behavior.

4.3.1 Non-MPI Routines

A breakdown of the amount of time spent locally on a compute node is shown in Fig. 12 as log-log plots, scaling w.r.t. the number of cores used. Only the top few most time consuming routines are considered.

If we remove the routines `wait_threads`() and `start_threads`() from the plots, we see a similar behavior by all the other routines as we saw in the previous configurations. The routine `init_pure`() again shows a constant time consumption irrespective of the number of cores, and becomes the dominant routine beyond 16 cores and 64 cores for the cases with `cc-pVQZ` and `cc-pV5Z` basis sets

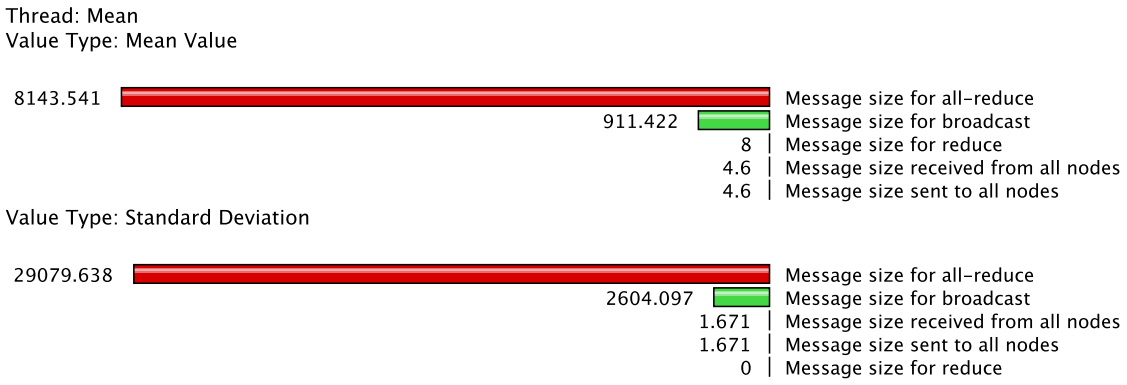

Figure 9: The mean (top) message sizes and their standard deviation (bottom) for various MPI routines in configuration MS while using 8 cores. There is a high variation in message sizes for all-reduce and broadcast.

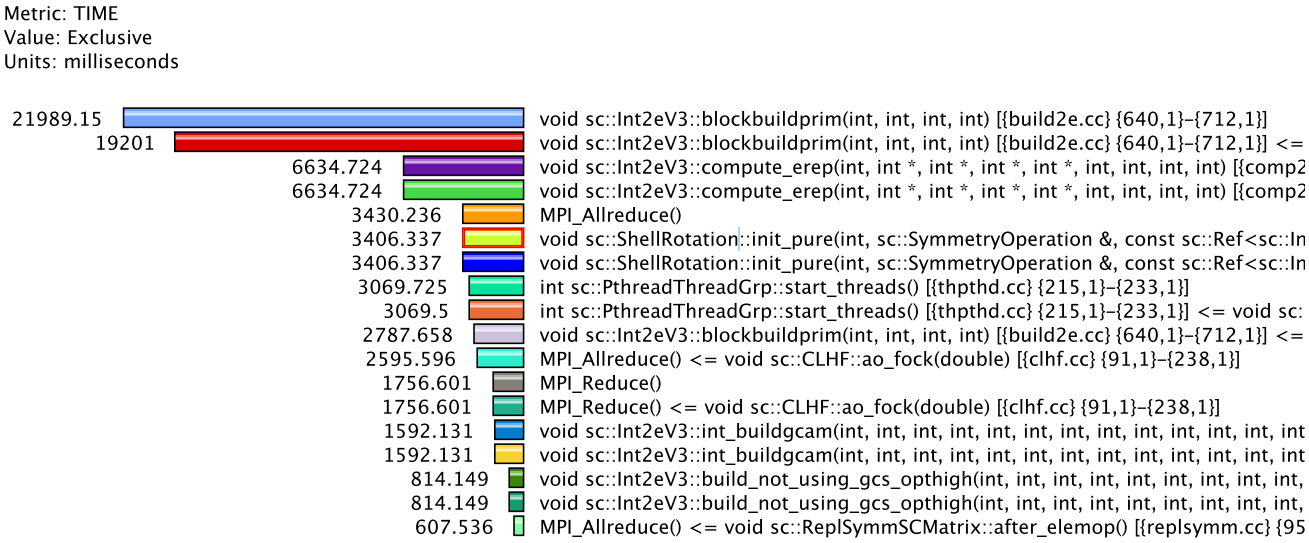

Figure 10: Mean exclusive execution times consumed by top few routines including both MPI and non-MPI routines when 8 cores are used.

respectively. Routines `block_build_prim`() and `compute_erep`() remain the most time consuming for most of the cases.

Since this is a threaded configuration, the thread routines play a major role in the execution times. In the plots in Fig. 12 we can clearly see that the routine `wait_threads`() is highest in terms of time consumption and dominates the overall execution time. This routine shows a significant slowdown beyond 64 cores in both cases above. This dictates the overall execution time to also show a slowdown.

4.3.2 MPI Routines

Fig. 13 contains log-log plots of a breakdown of the total time spent in the most time consuming MPI routines for the two cases of using basis sets $\texttt{cc}-\texttt{pVQZ}$ and $\texttt{cc}-\texttt{pV5Z}$.

As seen Fig. 13, the total time consumed by most MPI routines increases with the number of cores. The total MPI time is again dominated by the two routines, `MPI_Allreduce`() and `MPI_Bcast`(). We also see `MPI_Reduce`() routine to contribute significantly to the total time, although it shows

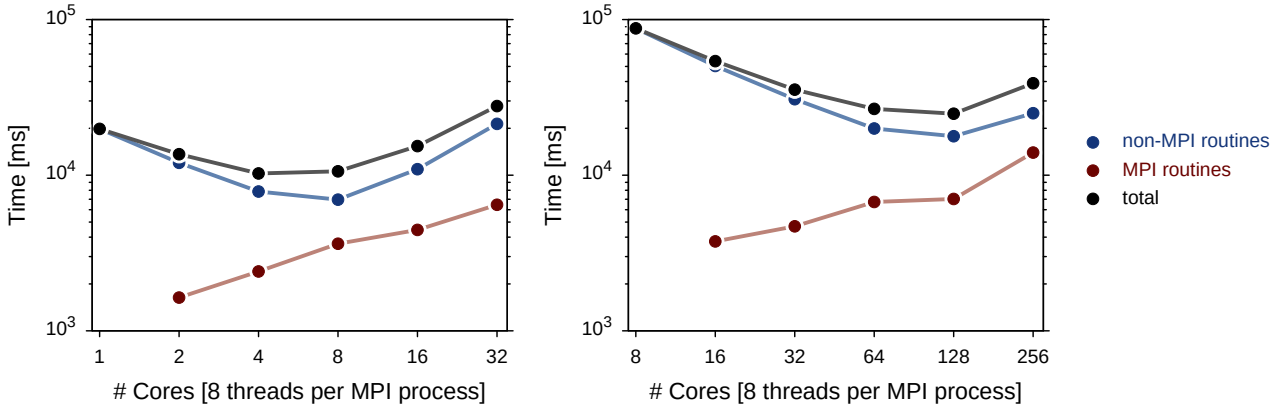

Figure 11: Configuration SM: 1 MPI processes per compute node, 8 threads per MPI process. Strong scaling w.r.t. number of cores used, of MPQC computing HF energies of H_2O. The plot on the left was obtained with the basis set `cc-pVQZ`, while the one on the right was with `cc-pV5Z`. In this configuration also all cores of a node are utilized.

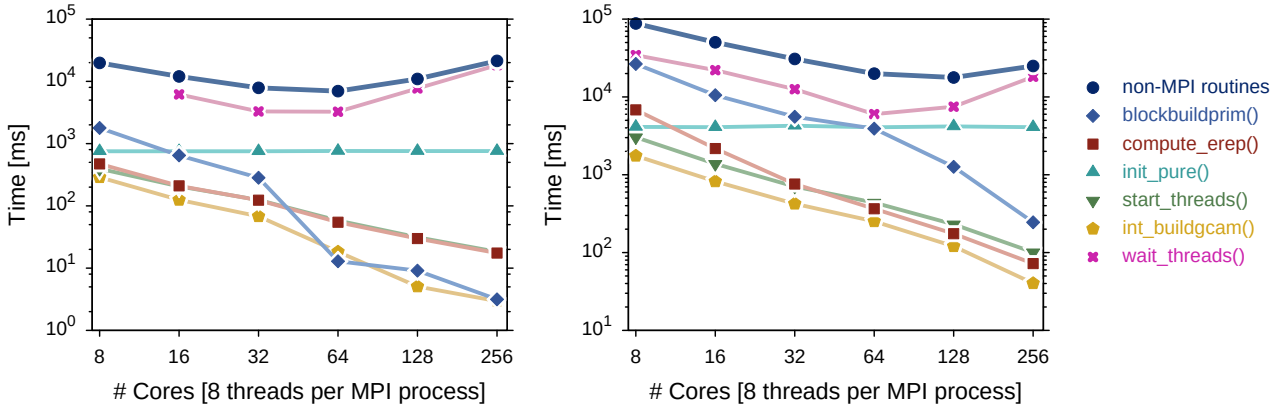

Figure 12: Breakdown of the time spent in non-MPI routines into times consumed by each individual routine in the configuration SM. On the left is a plot for execution using the basis set `cc-pVQZ`, and on the right is using `cc-pV5Z` basis set.

fluctuations around a flat line in the second case with `cc-pV5Z` basis set.

In Fig. 14, which shows the mean exclusive time consumed by various routines on 8 nodes, we see that the high mean value for `wait_threads`() routine skews the performance quite a lot. We also saw that there is a high imbalance in time consumed by various threads across the cores used. For example, a histogram in Fig. 15 shows that while most threads follow the normal distribution of time consumed, there are a few outliers which would greatly skew the performance.

4.4 Comparisons

We have now seen the performance of MPQC with the three different configurations, and also noted that `block_build_prim`() and `compute_erep`() are two routines which consume largest amount of time in most cases. The MPI communications and thread routines prove to be a bottleneck for larger number of cores. In this subsection we compare the results presented above for the three configurations in order to find which configuration works the best overall.

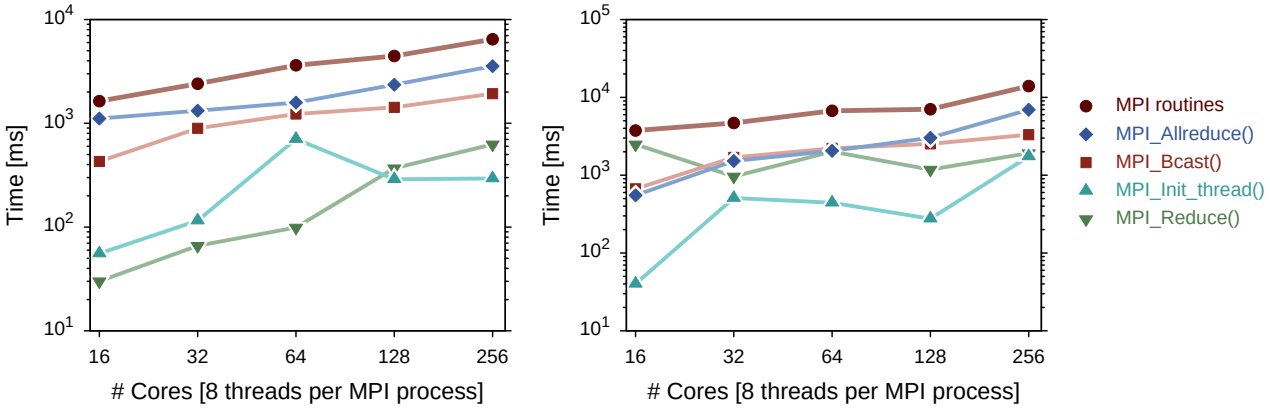

Figure 13: Breakdown of the total execution time spent in MPI routines into times consumed by each of the individual routines in configuration SM. On the left is a plot obtained using the smaller basis set `cc-pVQZ`, and on the right is one obtained using the larger basis set `cc-pV5Z`.

The plots above in Fig. 16 put together the total runtime, MPI routines time, and non-MPI routines time for the three aforementioned configurations for a comparison. We observe a very similar behavior for the configurations SS and MS, which are both non-threaded versions. The total execution times of these two cases are almost the same, time consumed by non-MPI routines are also almost the same. Time spent in MPI routines is slightly higher in the configuration SS compared to MS. This is expected because for the same number of cores, the former configuration uses larger number of nodes, equal to the number of cores used, than the latter, where a new node is added for every 8th MPI process, and the inter-node communication latencies are higher than intra-node communication latencies.

Further, we see that the overall performance of SS and MS configurations is higher than the threaded configuration SM given the same number of cores. The slowdowns observed beyond 32 cores in the first case (`cc-pVQZ`) and 64 cores in the second case (`cc-pV5Z`) are, as previously mentioned, due to dominance of MPI routines in configurations SS and MS, and due to thread routines in configuration SM.

5 Function `compute_erep()`

Given the above performance analysis results of MPQC, we pick the configuration MS for any further experiments since it performs the best in most cases. We recall the two most time consuming non-MPI routines are `block_build_prim()` and `compute_erep()`. We pick the `compute_erep()` for further performance analysis and optimizations.

The routine `compute_erep()` computes the two-electron repulsion integrals for a shell-quartet and reorganizes them in memory with respect to the molecular orbitals. It takes as input a shell-quartet and angular momenta of the four shells, and outputs the computed integrals for this input. This function involves a substantial data movement, which is a major bottleneck in this routine. During a single run of MPQC, this routine is called multiple times. In our experiments with the aforementioned inputs, depending on their sizes the number of calls ranges from $O(10^4)$ to $O(10^6)$ approximately. Fig. 17 shows the total time taken by this routine on each of the cores used. It shows quite good load balancing.

This function contains a number of deep loop nests. The main loop nest that we are considering is the one executed for majority number of times compared to others. This set consists of four outer

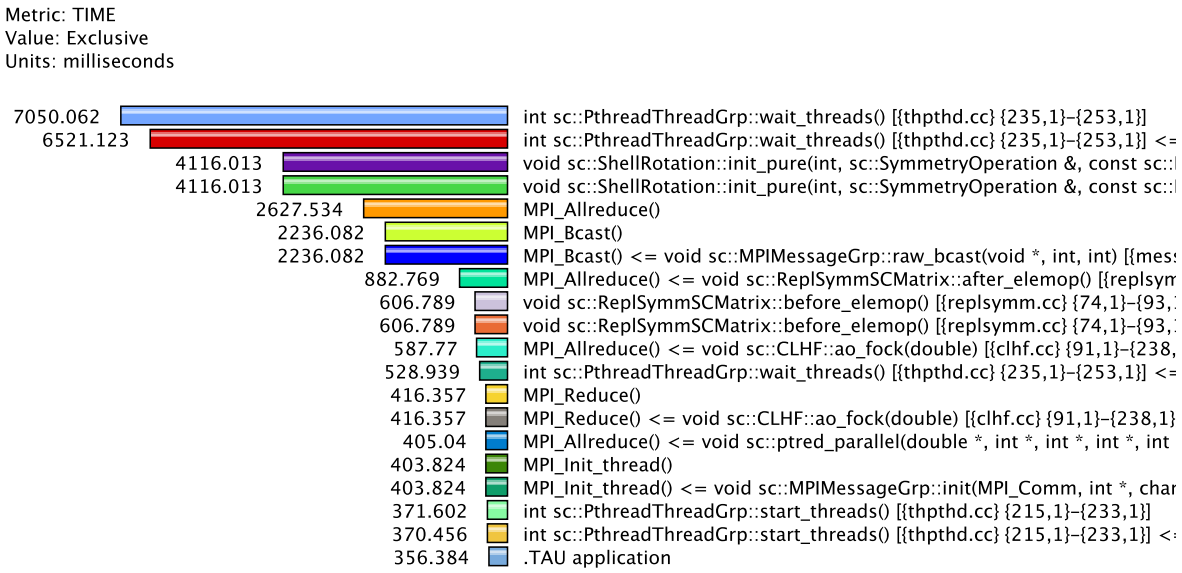

Figure 14: The mean exclusive times consumed by various routines when using 8 nodes, each with 8 threads (64 threads in total.)

loops, and five sets of inner loops under some conditions, each set consisting of three to four loops. The basic structure of the outer loop nest is shown in a selective code extract in Fig. 18. This set of the four outer loops is over the numbers of contractions in the shell-quartet under consideration. The index ranges of the inner loops are computed here as the number of cartesian functions in the shells of the shell-quartet computed using the respective angular momentum over their centers.

The outer loop nest contains five conditional inner loop nests – first, third and fifth nests with four loops each, and second and fourth with three loops each. The first loop nest's code extract is shown in Fig. 19. This nest is executed for the majority number of calls to `compute_erep` and, hence, we will mainly focus on this set in the rest of the following analysis and optimization. Loop nest four is the second in majority of the number of calls, and we will consider this for some parts in our analysis as well.

The main purpose of this loop nest 1 is to reorganize and move the computed integrals from `shiftbuffer` to `int_buffer`. This memory copy has sequential reads from `shiftbuffer`, but strided writes to `int_buffer`. These stride sizes are not constant, and are computed on the fly based on which cartesian function of which contraction number of each of the shells in the quartet are currently under consideration, which are defined by the above eight loops.

6 Intra-function Performance Profiles and Analysis

In the following we present some of the performance analysis results of our focal function `compute_erep`(). This part of the analysis is based on the hardware counters for various metrics. We used PAPI [10] and Intel Vtune [11] to obtain these performance profiles. PAPI gathers hardware counter data during an execution and supports parallel codes. It can also obtain loop level profiles, and provides execution time information but can obtain information from a limited set of the hardware counter based metrics. Vtune provides information from many of the available hardware counters. It performs multiple runs of the code to obtain the values from the specified counters, with runs consuming a lot more time compared to a typical execution of the code. Hence, Vtune is not used for time based analyses. We

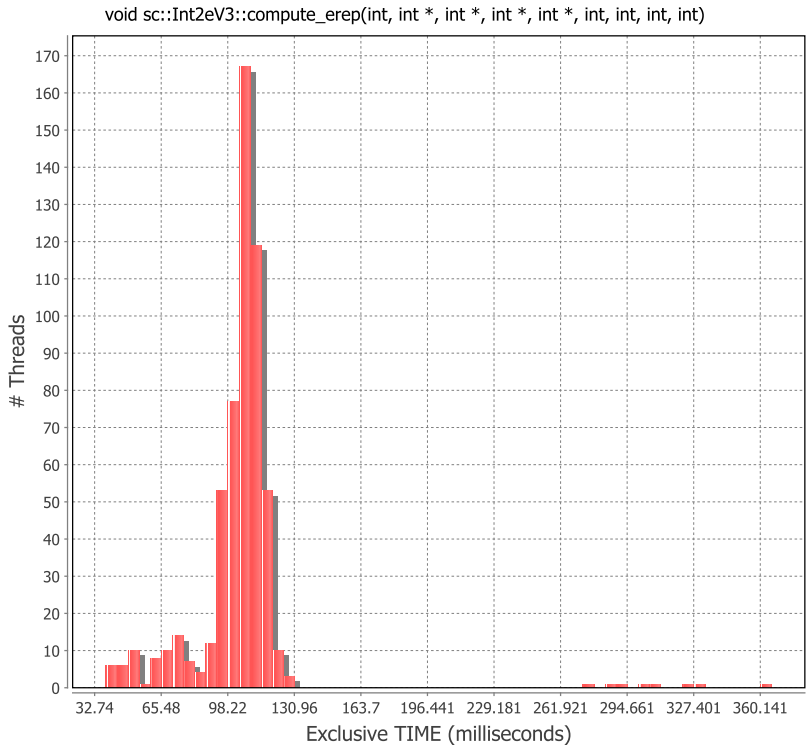

Figure 15: A histogram of the number of threads binned according to the amount of time consumed by each. Note the normal distribution of time w.r.t. number of threads, and a few outliers which have high values of time consumption. These outliers skew the performance by a large margin.

obtain profiling data for a number of hardware metrics. These metrics include data TLB load and store hits and misses, load and store hits and misses for various caches available (L1D, L2, LLC), number of instructions issued and retired, number of cycles, number of floating point operations, and the such. In the following we present performance scaling data for various inputs and number of processes.

6.1 Scaling with Number of Processes

In order to study the behavior of various metrics with respect to the number of processes, we obtained the data presented and discussed below. In Fig. 20 we show plots of the obtained data. Plot (a) shows the scaling of execution time of `compute_erep`(), its main outermost loop and the first and fourth inner loop nests as a function of the number of processes. The obtained results are quite as expected, and we see almost perfect scaling. This is mainly because `compute_erep`() does not contain any MPI communication within. Plot (b) shows the number of floating point operations per process and (c) shows the corresponding Flops. Again, the number of floating point operations scale as expected and the obtained Flops values obtained remain near constant. Plots (d) and (e) in Fig. 20 show scaling data on instructions per cycle (IPC) and percentage of instruction replay respectively. In both cases we obtain nearly the same values, irrespective of the number of processors – again, as expected. The expected behavior of all these metrics basically means that the function `compute_erep`() computationally scales well with the number of processes.

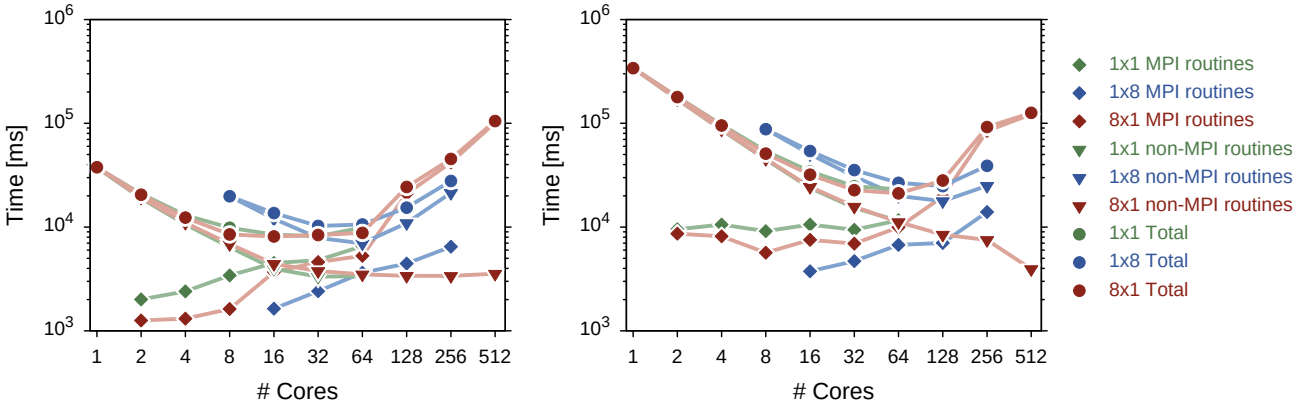

Figure 16: Comparison of the three configurations SS, MS and SM. On the left is a log-log plot for the case with `cc − pVQZ` basis set and on the right is one with `cc − pV5Z` basis set. These plots show the total execution times, execution times for non-MPI routines, and for MPI routines in the three configurations for comparison. Blue lines represent configuration SS, green represents configuration MS and red represents configuration SM.

6.2 Scaling with Input Molecule Size

We further study the performance of `compute_erep` with respect to the input molecules. For this, we use the molecules H_2O (water), H_4O_2 (water dimer), CH_4 (methane), C_4H_{10} (butane), and C_8H_{18} (octane) as described in Section 2. Fig. 21 contains plots of the obtained results. Similar to the previous scaling results, we obtain data on the same metrics. Fig. 21 (a) shows the execution times of `compute_erep`(), and the main loop and inner loop nests 1 and 4, for the various input molecules. Except for the dip for H_2O we see that the times scale almost linearly with the molecular mass of the molecules. Plot (b) shows the number of floating point operations and (c) shows the performance in Flops. IPC and instruction replay rate are shown in plots (d) and (e) respectively. We note that IPC tends to increase slightly with the molecule size, as does the overall performance (Flops).

7 MPQC Optimization

Based on all the above analysis, we study the optimization of the function `compute_erep`(). In the following we perform a series of optimizations, from loop optimization to memory movement optimizations, and study the performance change compared to the original code.

7.1 Loop Optimizations

A few loop optimization techniques were explored, and loop unrolling proved to be the most beneficial in terms of performance. We unrolled the innermost loops of the loop nests 1 and 4 by a factor of 2, 4 and 8 to analyze the performance improvements. Fig. 22 shows some performance data comparing the original code with loop unrolled code, with unroll factors of 2 and 4. We see that in most cases, unrolling the loops by factor of 2 improves performance the most, and we use this for all subsequent experiments. The unroll factor is a parameter which can be tuned based on the system to obtain best performance.

In Fig. 23 we show the comparison in performance between the original MPQC code, and code with loop optimization (unrolling). Data is shown for different input molecules. In the plots, the

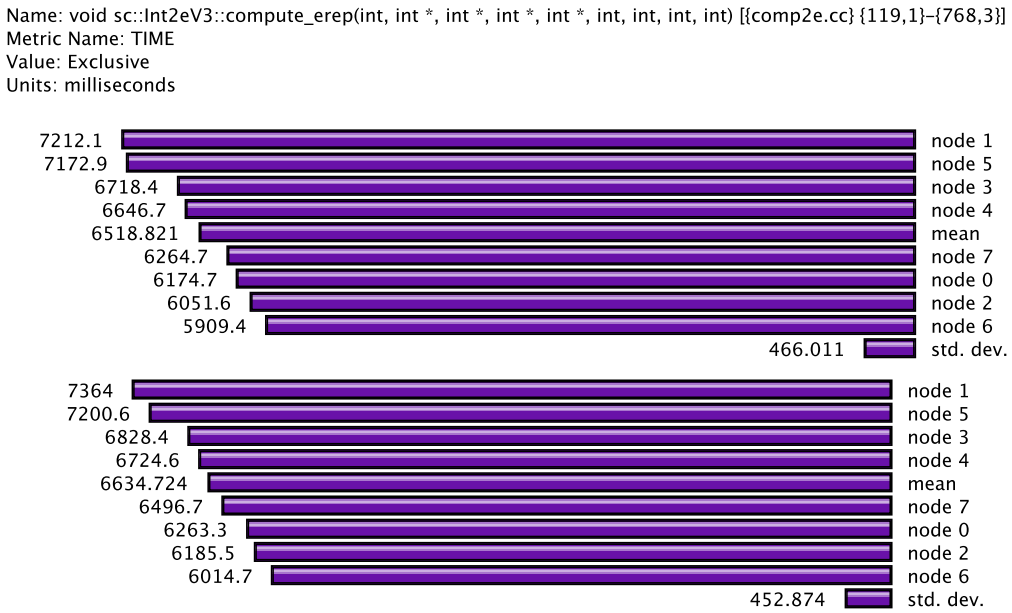

Figure 17: The distribution of time taken by the routine `compute_erep`() across all the 8 cores used along with the means and standard deviations. The top chart is with configuration SS and bottom is with configuration MS.

loop optimized version of the code shows about 18% decrease in the execution time for inner loop 1 and 9% for inner loop 4. It also has significant decrease in the number of instruction replays – 41% less instruction replays for inner loop 1 and 33% for inner loop 4. Due to the observed decrease in execution time with the optimized code, Flops achieved by the code shows an increase with 10% for inner loop 1 and 22% for inner loop 4. Further, the overall average number of instructions executed per cycle also shows an increase of 32% for inner loop 1 and 4% for inner loop 4.

7.2 Memory Access Optimizations

We performed a detailed memory profiling for the various components of the function `compute_erep`(). This includes the memory access patterns, amount of cache hits and misses and TLB hits and misses. As we mentioned earlier, memory reads in loop nest 1 are sequential, but memory writes are not contiguous. The stride values between consecutive writes varies with the loop iterations. Fig. 24 shows a histogram of the values of strides. It shows that smaller stride values occur more frequently than higher values, as well as the number of smaller stride values are larger than higher values. A strided memory access pattern causes increased cache and TLB misses. Given the above pattern, in order to optimize the memory writes, we implemented a data blocking scheme. It uses a bucketing approach based on the actual index in the destination buffer. An intermediate buffer small enough to fit in the cache is used to temporarily store the integral values being written, and once the buffer is full, the integral values are written to their intended destination indices in a sequential manner. This introduction of sequentiality tends to improve the performance by reducing the number of cache and TLB misses.

The size of the buckets (and the number of bits used to hash map indices) is a tunable parameter. After exploring a few values, we obtained best performance with number of bits as 8, and hence bucket size as 256. Using these values, we performed further performance analysis and comparison with previous version of MPQC code and the original version. In Fig. 25 we present such data for

```
// am1, am2, am3, am4 = angular momenta over shell centers
ogc1 = 0;
for (int i = 0; i < nc1; i++) {
  int tsize1 = INT_NCART_NN(am1);
  ogc2 = 0;
  for (int j = 0; j < nc2; j++) {
  int tsize2 = INT_NCART_NN(am2);
    ogc3 = 0;
    for (int k = 0; k < nc3; k++) {
      int tsize3 = INT_NCART_NN(am3);
      ogc4 = 0;
      for (int l = 0; l < nc4; l++) {
        int tsize4 = INT_NCART_NN(am4);
        // ...
        // five conditional inner loop nests ...
        // ...
        ogc4 += tsize4;
      }
      ogc3 += tsize3;
    }
    ogc2 += tsize2;
  }
  ogc1 += tsize1;
}
```

Figure 18: The main outer loop nest structure in the function `compute_erep`(). This set consists of four nested loops.

various performance metrics. From these results we observe significant performance improvement with the combined loop and memory access optimizations.

8 Conclusions

With the target of analyzing and optimizing MPQC code, we performed loop and memory access optimizations within the function `compute_erep`(). The improvement in the overall execution time for the application, as well as for `compute_erep`() and its inner loop nest 1 is shown in Fig. 26. For a reference point, we use contiguous block copy in place of the required memory movements, also shown in Fig. 26 and compare the obtained performance with the ideal performance. The performance of the optimized code turns out to be quite close to the ideal situation.

9 Acknowledgement

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] "Massively Parallel Quantum Chemistry," Sep. 2012. [Online]. Available: {http://www.mpqc.org}

```
int red_index = 0;
int indexoffset = F(ogc1, ogc2, ogc3, ogc4);
int index1 = indexoffset;
for (int ci1 = 0; ci1 < tsize1; ci1++) {
  int index12 = index1;
  for (int ci2 = 0; ci2 < tsize2; ci2++) {
    int index123 = index12;
    for (int ci3 = 0; ci3 < tsize3; ci3++) {
      int index1234 = index123;
      for (int ci4 = 0; ci4 < tsize4; ci4++) {
        int_buffer[index1234] = shiftbuffer[red_index + ci4];
        index1234 += stride4;
      }
      red_index += tsize4;
      index123 += stride3;
    }
    index12 += stride2;
  }
  index1 += stride1;
}
```

Figure 19: The structure of one of the five inner loop nests, contained within the outer loop nest shown in Fig. 18 in function `compute_erep`(). This set of nested loops is over the number of cartesian functions in each shell of the shell-quartet. The main operation in this loop nest is memory copy.

[2] C. L. Janssen and I. M. B. Nielsen, *Parallel Computing in Quantum Chemistry*. CRC Press, Apr. 2008.

[3] ——, "Second-Order Moller-Plesset Perturbation Theory," in *Parallel Computing in Quantum Chemistry*. CRC Press, Apr. 2008, pp. 147–166.

[4] C. Møller and M. S. Plesset, "Note on an Approximation Treatment for Many-Electron Systems," *Physical Review*, vol. 46, pp. 618–622, Oct 1934.

[5] C. L. Janssen and I. M. B. Nielsen, "Two-Electron Integral Evaluations," in *Parallel Computing in Quantum Chemistry*. CRC Press, Apr. 2008, pp. 117–130.

[6] P. M. W. Gill, *Molecular Integrals Over Gaussian Basis Functions*, ser. Advances in Quantum Chemistry. Academic Press, Inc., 1994, vol. 25.

[7] J. T. Fermann and E. F. Valeev, "Fundamentals of Molecular Integrals Evaluation," Tech. Rep.

[8] "MPQC GaussianBasisSet Class Reference," Sep. 2012. [Online]. Available: {http://www.mpqc.org/mpqc-html/classsc_1_1GaussianBasisSet.html}

[9] S. S. Shende, "The Tau Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, May 2006. [Online]. Available: http://hpc.sagepub.com/cgi/doi/10.1177/1094342006064482

[10] Innovative Computing Laboratory (ICL), University of Tennessee, "Performance Application Programming Interface (PAPI)," Sep. 2012. [Online]. Available: {http://icl.cs.utk.edu/papi/index.html}

[11] Intel Corporation, "Intel VTune Amplifier XE 2013," Sep. 2012. [Online]. Available: {http://software.intel.com/en-us/intel-vtune-amplifier-xe}

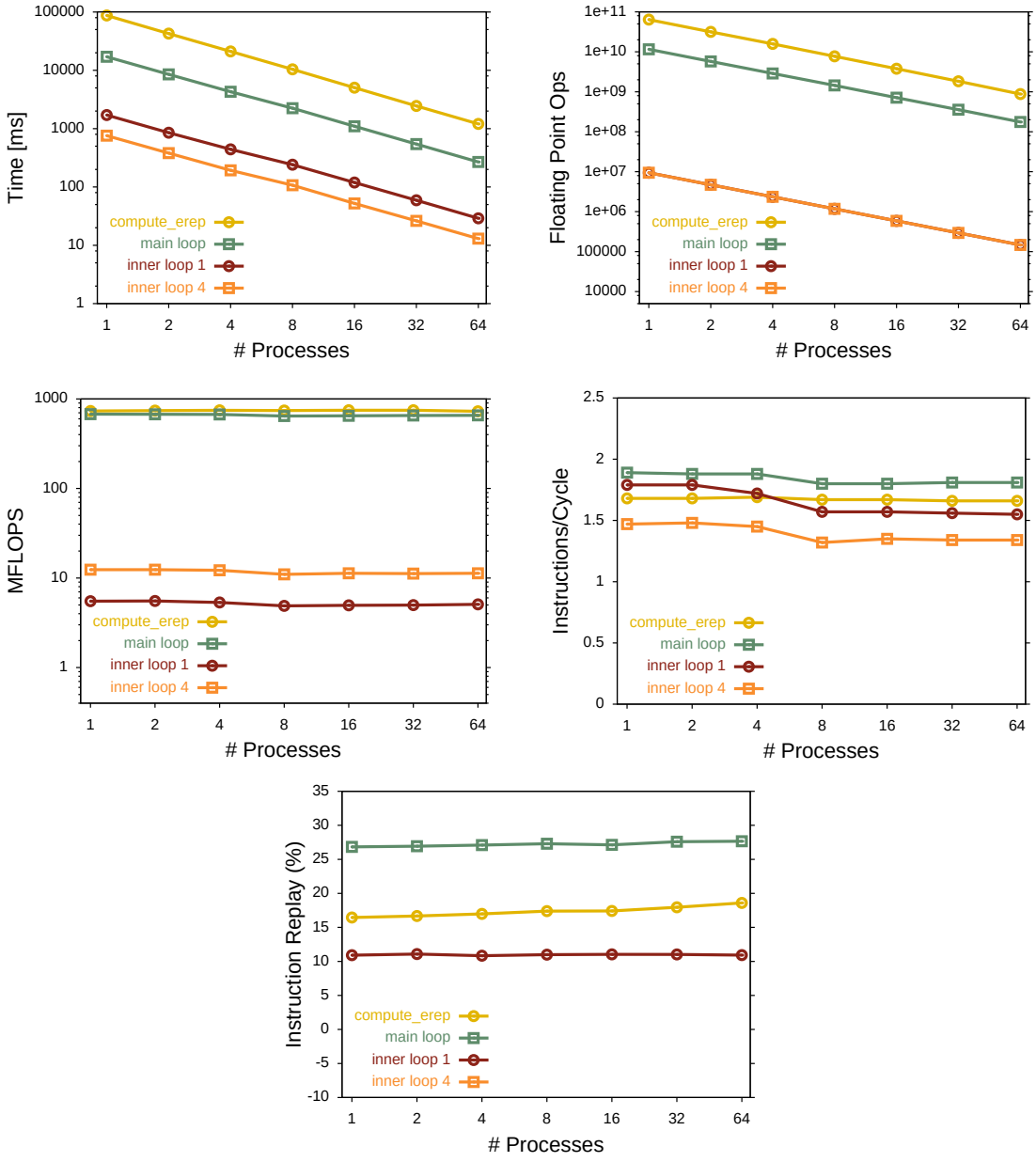

Figure 20: Scaling of various metric values with respect to the number of processes: (a) execution times, (b) number of floating point operations, (c) performance as floating point operations per second, (d) average number of instructions executed per cycle, and (e) percentage of instructions replayed. The total values for `compute_erep`(), and its components, the main outermost loop and first and fourth inner loop nests, are shown.

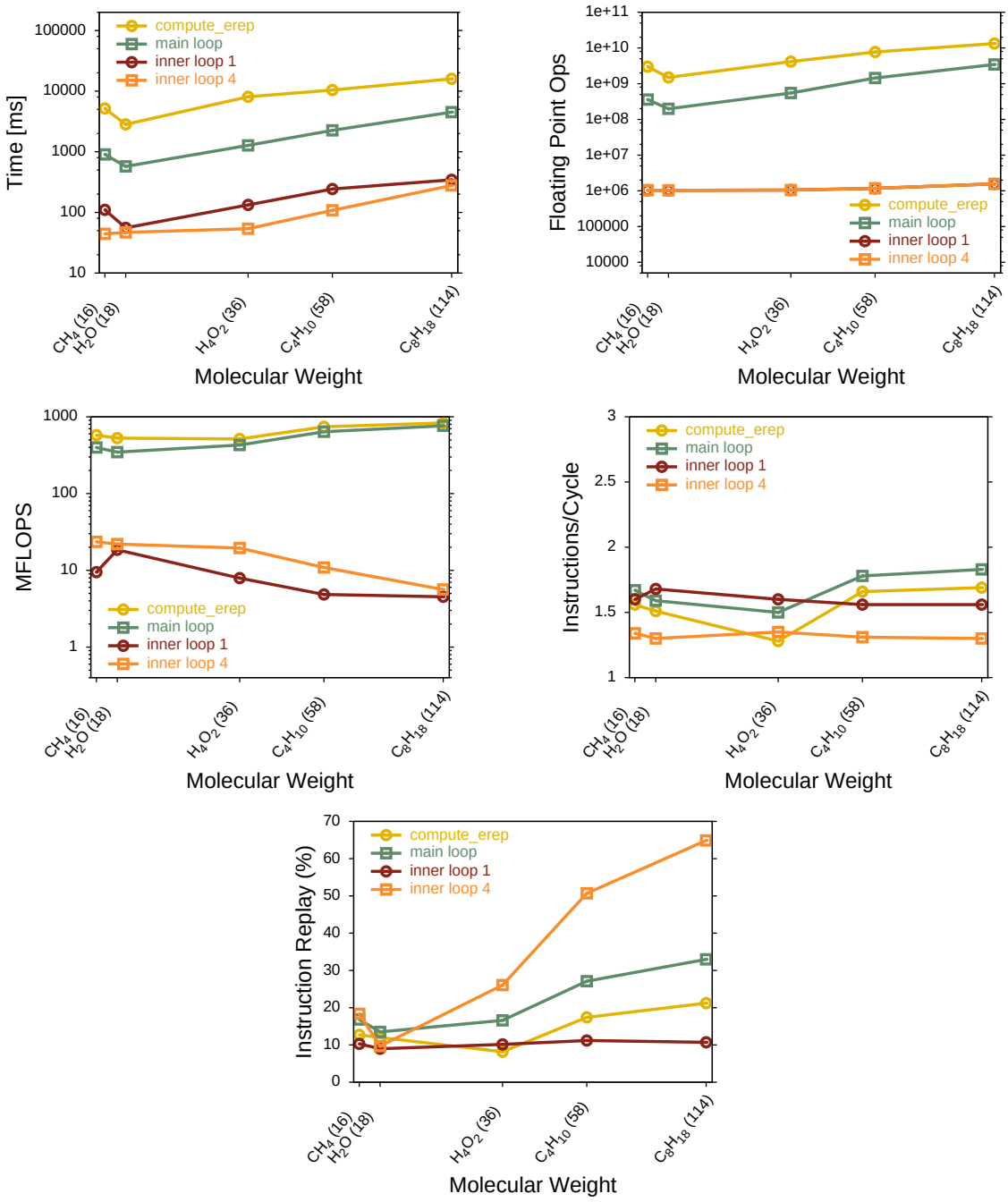

Figure 21: Various performance metric values with respect to the molecular mass of the input molecule: (a) execution time, (b) number of floating point operations, (c) performance in floating point operations per second, (d) overall average number of instructions executed per cycle, and (e) instruction replay rate. Values for `compute_erep`() and its components are shown.

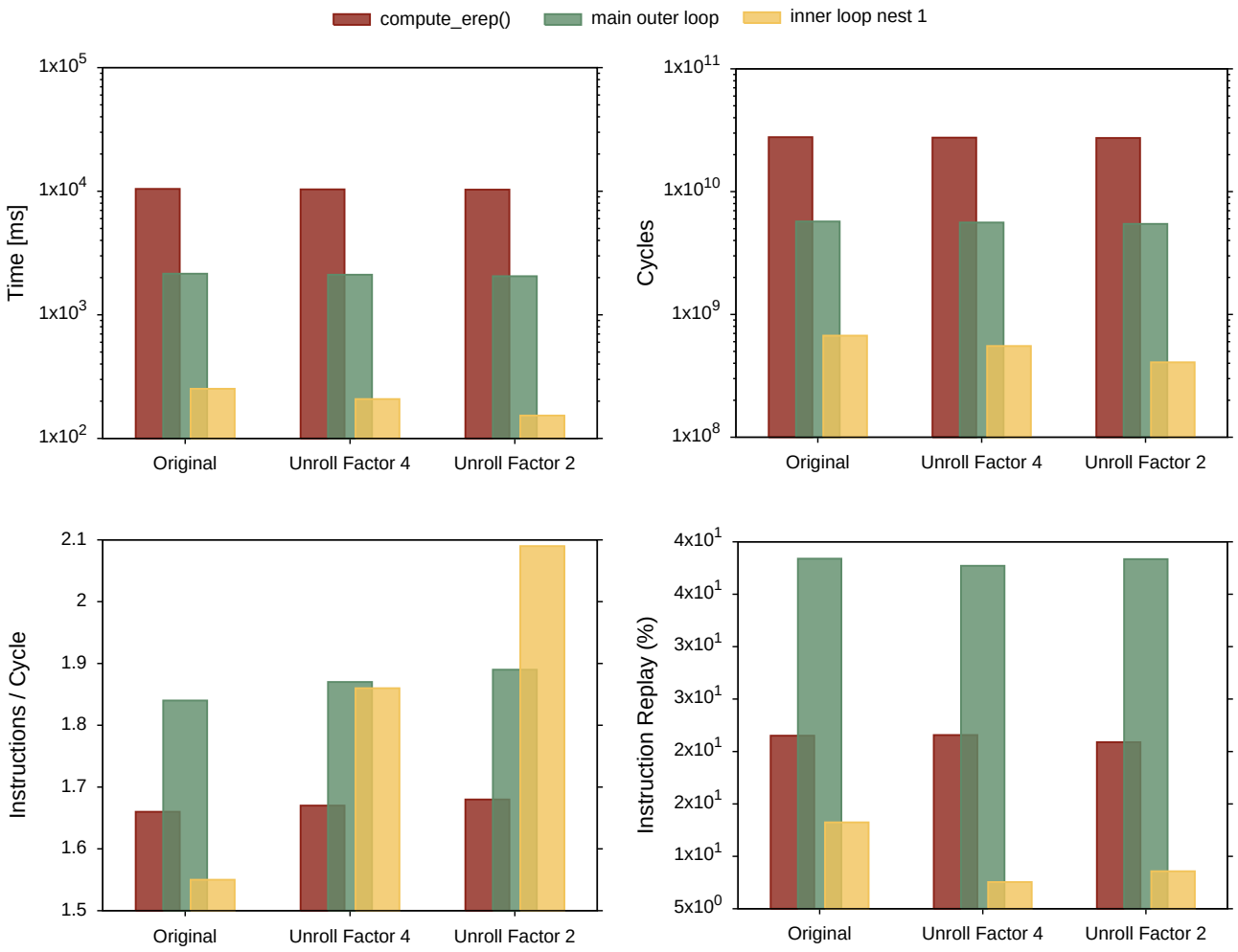

Figure 22: Performance of `compute_erep`() and its component loops with different loop unroll factors: (a) execution times, (b) total number of cycles, (c) average number of instructions executed per cycles (IPC), and (d) instruction replay rate.

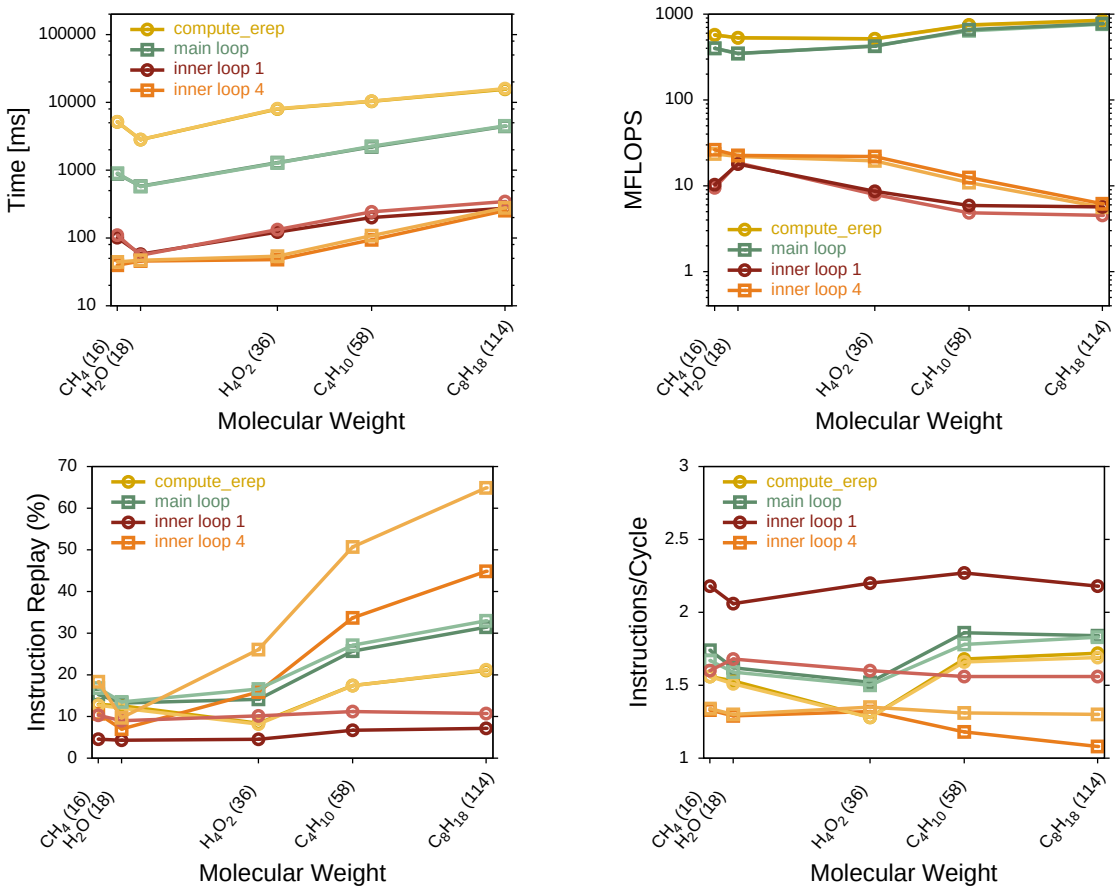

Figure 23: Performance comparison between original MPQC code (light shades) and the loop optimized code (darker shades): (a) execution times, (b) performance as floating point operations per second, (c) instruction replay rate, and (d) average number of instructions per cycle. Performance is shown for the function `compute_erep`, its main outer loop, and two of its inner loops.

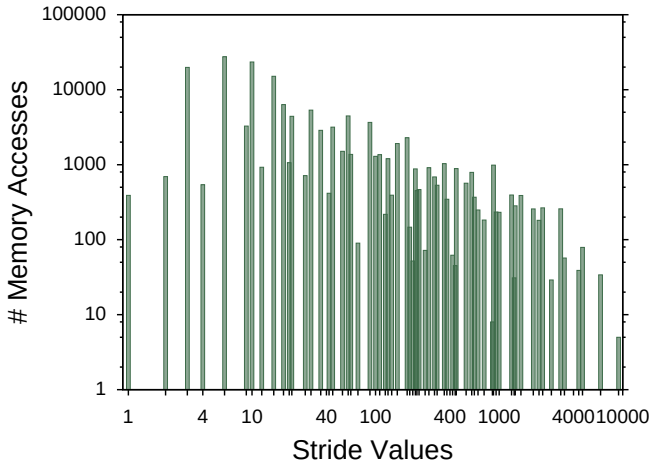

Figure 24: A histogram showing the number of memory accesses w.r.t. the stride values of array indices between consecutive writes to the destination integral buffer `int_buffer[]`.

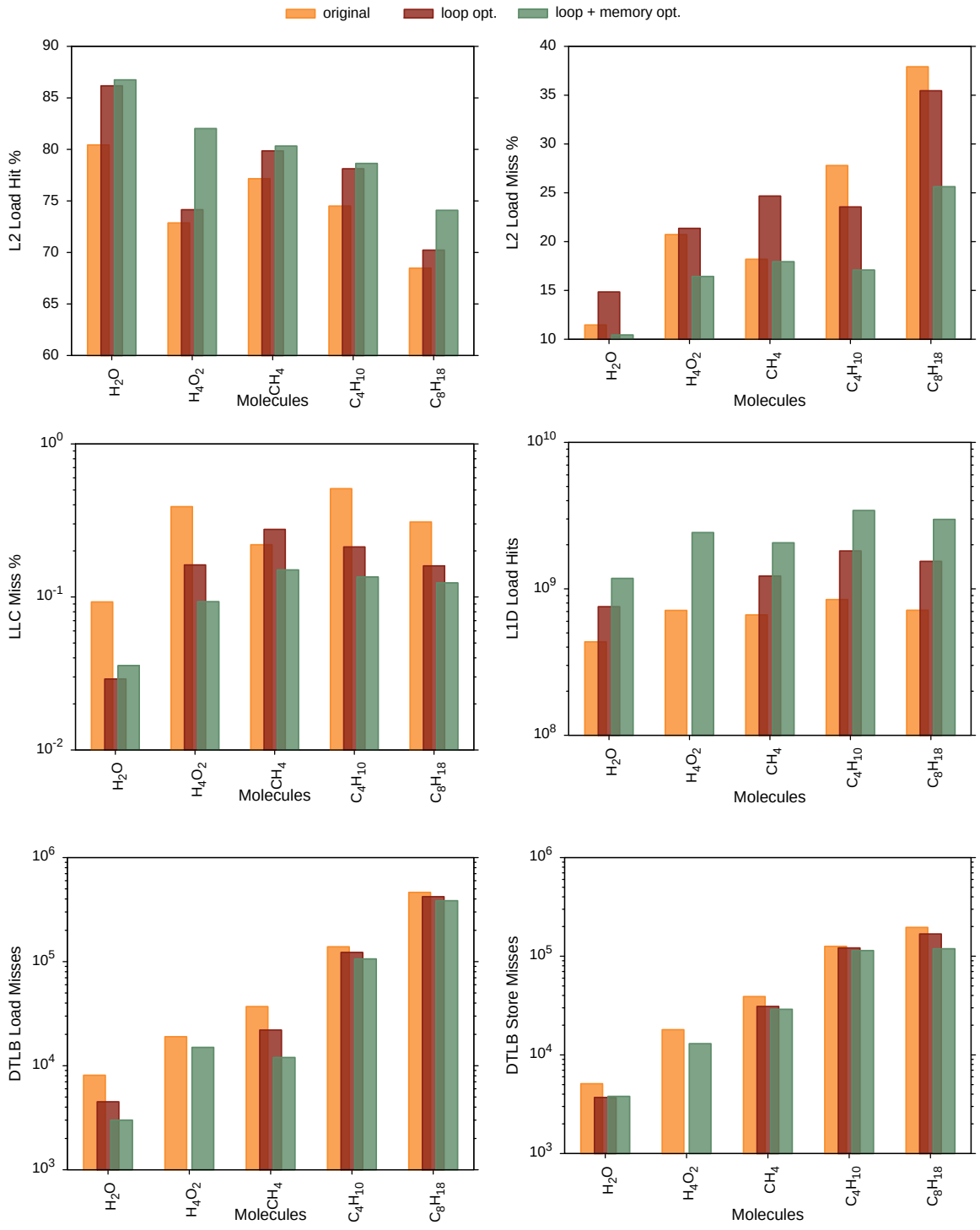

Figure 25: Performance comparison between the original MPQC code (original), loop optimized code (loops-opt), and memory movement optimized code (final): (a) L2 cache load hits, (b) L2 cache load miss, (c) LLC (L3 cache) miss, (d) data L1 cache hits, (e) data TLB load misses, and (f) data TLB store misses.

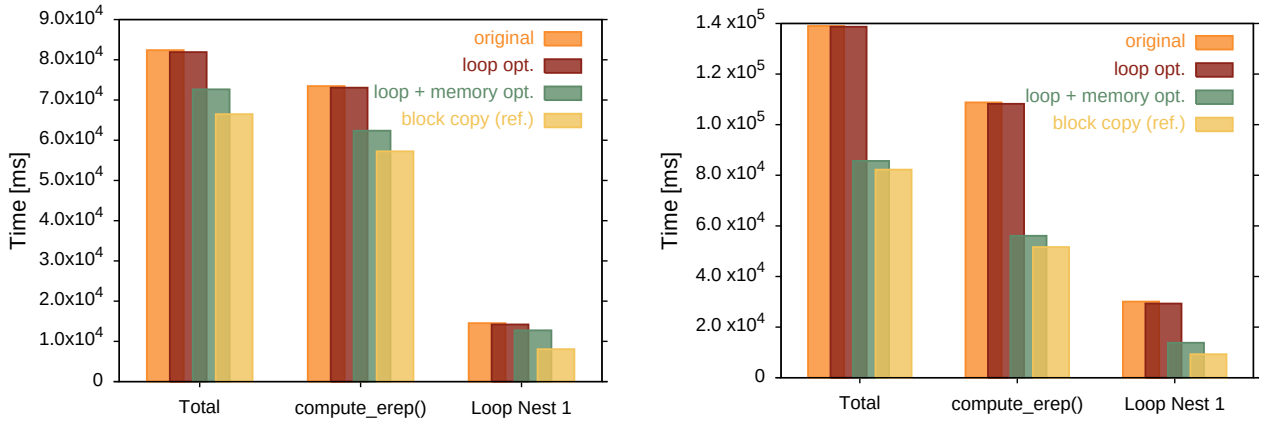

Figure 26: Performance comparison between the original MPQC code (original), loop unroll optimized code (loop unrolling), and final optimized version consisting of loop optimization and memory write optimizations (bucketing+unrolling). Data is shown for input molecules C_4H_{10} (left) and C_8H_{18} (right). For a reference, a contiguous block copy in the place of actual data movement is also shown.