Computer Physics Communications 185 (2014) 3175–3188

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

# Shared-memory parallelization of a local correlation multi-reference CI program

Johannes M. Dieterich [a], David B. Krisiloff [b], Alexander Gaenko [c,d], Florian Libisch [a], Theresa L. Windus [c,d], Mark S. Gordon [c,d], Emily A. Carter [a,e,*]

[a] Department of Mechanical and Aerospace Engineering, Princeton University, Princeton, NJ, 08544-5263, USA

[b] Department of Chemistry, Princeton University, Princeton, NJ, 08544-1009, USA

[c] Department of Chemistry, Iowa State University, Ames, IA 50011, USA

[d] Ames Laboratory, Ames, IA 50011, USA

[e] Program in Applied and Computational Mathematics, and Andlinger Center for Energy and the Environment, Princeton University, Princeton, NJ, 08544-5263, USA

ARTICLE INFO

*Article history:*
Received 1 June 2014
Accepted 19 August 2014
Available online 27 August 2014

*Keywords:*
Local correlation
Parallelization
Shared memory
Multi reference
Dioxirane
Multi-reference configuration interaction

ABSTRACT

We present a shared-memory parallelization of our open-source, local correlation multi-reference framework, TigerCI. Benchmarks of the total parallel speedup show a reasonable scaling for typical modern computing system setups. The efficient use of available computing resources will extend simulations on this high level of theory into a new size regime. We demonstrate our framework using local-correlation multireference computations of alkyl-substituted dioxirane and solvated methyl nitrene as examples.



## 1. Introduction

Moore's law [1], promising an exponential growth of transistor counts, has been the metronome of the chip industry for the last five decades.[1] While higher transistor counts for a long time directly increased single thread performance, in the last decade the additional transistors were used to herald the advent of multi-core processors and processing. This development bridged the gap between single-core and massively parallel systems, effectively rendering single-core computing obsolete.

Computational chemistry as a high performance computing (HPC) discipline transforms computational resources into the science of increasingly complex and larger chemical systems and reactions. Within computational chemistry, one can observe wide usage of correlated wavefunction methods due to their high accuracy. As the systems that are studied become more complex, frequently their multiconfigurational character becomes more dominant, ergo a need for efficient multireference correlated wavefunction methods arises. Efficient formulations and implementations of multireference coupled cluster (CC) and configuration interaction (CI) methods exist [2–5]. However, these (semi)canonical implementations share a prohibitively high scaling with the molecular size. Even the most efficient parallelization is unsuitable to overcome such intrinsic algorithmic bottlenecks.

A standard tool in single-reference correlated wavefunction methods has been to exploit the local nature of electron correlation to achieve reduced scaling compared to the canonical equivalent [6–11]. Recently, efficiently parallelized from-scratch implementations of local single-reference methods have been reported [12]. No parallel implementation of reduced scaling multireference correlated wavefunction codes exists to date.

In this contribution, we report the implementation of a shared-memory parallelization for our reduced scaling code, TigerCI [13–20], implementing a Cholesky-decomposed multi-reference local correlation algorithm. This will enable TigerCI to exploit modern shared-memory hardware. The algorithmic protocol of TigerCI has been reviewed at length in Ref. [20]; we therefore only

* Corresponding author at: Department of Mechanical and Aerospace Engineering, Princeton University, Princeton, NJ, 08544-5263, USA. Tel.: +1 609 258 5391; fax: +1 609 258 5877.

*E-mail address:* eac@princeton.edu (E.A. Carter).

[1] Interestingly, Gordon Moore himself had a background in Chemistry, not Engineering.

http://dx.doi.org/10.1016/j.cpc.2014.08.016
0010-4655/

concisely summarize background theory and implementation in this contribution where needed.

We motivate which parallelization framework was used in this work, explain the development principles followed in parallelizing sections of the code and assessing both the parallel speedup of those individual sections and the overall code. On a side note, we wish to elucidate some programming practices which facilitate maintaining and enhancing existing Fortran code bases.

As an outlook, we discuss how exploiting available computing resources in conjunction with a local correlation multireference wavefunction framework allows routine simulation of large systems at this high level of theory and present local-correlation multireference computations of alkyl-substituted dioxirane and solvated methyl nitrene as examples.

## 2. Methodology

We describe herein the parallelization of our existing TigerCI [13–15,17–21] Multi-Reference Singles and Doubles Configuration Interaction (MRSDCI)/Averaged Coupled Pair Functional (MRACPF) implementation. The challenges encountered and subsequent strategies developed in this task are not unique. Indeed, we believe they apply to many legacy code bases in the field of computational chemistry. Due to the complexity of the existing code, invasive changes and/or rewrites typically have to be kept to a minimum. Additionally, the serial code paths are typically well-tested, mature, and in production. Therefore, new features such as parallelization should be made to integrate well with and reuse if possible the existing code paths. Finally, efforts towards parallelization have to be balanced with code maintenance and optimization. In particular, the constant shift in hardware capabilities and bottlenecks requires updates in the details of the implementation. Consequently, having a robust parallelization framework that avoids code duplication is essential.

Many frameworks exist to parallelize program code execution. For HPC applications, the most commonly used ones are OpenMP [22], MPI [23], GDDI [24], and global arrays [25]. Numerous additional layers abstracting or wrapping these frameworks exist, perhaps most notably MPI wrappers for other languages than C and Fortran (e.g., Refs. [26,27]). In the case of TigerCI, more than 90% of the code base is Fortran 95. The only other relevant code part is written in C++, simplifying the choices to the OpenMP, MPI, GDDI, and global arrays ones.

When selecting one of those frameworks, the central question is which one can satisfy the expected application scalability while minimizing overhead for the application in question. Overhead can mean both development overhead and overhead at runtime. Runtime parallel overhead involves a constant (related to the number of processors) contribution due to the additional complexity of a parallel code path, as well as communications overhead that will (possibly super-linearly) increase with the number of cores and/or the problem size. For massively-parallel applications assumed to scale well up to hundreds or thousands of CPUs, both the programming overhead and the constant overhead, i.e, the question of whether the parallel algorithm is slower in serial mode than the best serial algorithm are of little concern. MPI caters to the needs of such codes. On the other hand, for tasks assumed to scale only up to a small number of CPUs, overhead in all its forms is a much more pressing problem. Added runtime overhead will likely annihilate part of the parallelization benefits while a significant programming overhead may render parallelization economically unfeasible. This is even more true if the application is not (entirely) CPU-bound but memory- or storage-bound. Optimal usage of these resources must then be ensured.

OpenMP is an advanced programming interface (API) specification for parallel programming which can be used for codes that would scale poorly with many cores due to communications overhead. Its low programming overhead stems from the high abstraction it provides (no explicit thread handling is needed) and the elegant, directive/pragma-based API. Each parallel region is enclosed with preprocessor instructions for an OpenMP-enabled compiler. Ideally, this allows one to add parallelization to existing code with little programming overhead and maximum reuse of existing, efficient serial code paths.

We expect the scalability of the TigerCI application to be modest. Although classic publications indicate the possibility of implementing non-local CI algorithms in a massively parallel fashion [4], one should keep in mind that computing hardware has significantly evolved in the last two decades, giving rise not only to exponentially more powerful cores but also putting even more stress on serial bottlenecks such as communication and I/O. Additionally, the TigerCI code has, unlike the code described in Ref. [4], not been designed with parallel scalability in mind but instead for maximal serial throughput: the reduction of floating point operations was the prime target in the design of the symmetric group graphical approach (SGGA) engine [28–32] that is at the heart of the TigerCI code.

For the reasons discussed above, this parallelization work will be based on the OpenMP API. The natural way to use OpenMP is for a loop-based parallelization, i.e., evaluating the loop body concurrently for different loop indices. The success of this strategy depends on two factors: how decoupled the loop body is in between different indices and how much computational workload exists within the loop body compared to outside of it. Therefore, it is of utmost importance to identify those parts of the algorithm that satisfy these requirements. Naturally, this can be accomplished with a hotspot analysis as reported in Ref. [20]. According to the Pareto principle as conceived by Juran [33], in any system there are a *vital few and trivial many*. This observation is commonly applied to computer codes because only a fraction of the code (typically 20%) is responsible for the vast majority of the computing time (typically 80%). Hence, another common name of the Pareto principle is the 80–20 rule. Therefore, parallelization of said fraction will suffice to obtain some parallel speedup. Amdahl's law [34] can be used as an approximation for the ideal parallel speedup. In our case, we aim to parallelize in excess of 90% of the typical wall time. This parallelized fraction would, following Amdahl's law, ideally cause an asymptotic speedup of 10. As this approximation does not take into account serialization points within those parallelized kernels, we expect a smaller speedup in practice.

Following the identification of the relevant kernels/loops to be parallelized, some preparation of said kernel is typically required. This preparation includes loop reordering in order to maximize the weight of the loop body while keeping individual bodies decoupled. However, in the context of legacy codes, the most important step in decoupling loop iterations is the removal of shared global states. Unfortunately, most legacy Fortran codes use global states either in the form of common blocks or global module variables for information persistence and/or communication. Obviously, if this strategy is used within the target kernels, parallelization will be cumbersome to impossible. Therefore, transformation of global states into a functional programming style, i.e., mutable states in subroutine parameters, is mandatory.

The remaining challenge is locking, which sanitizes the concurrent manipulation of shared data from different threads. In the context of our local CI code, locking is most important for the sigma vector (product of the Hamiltonian and the CI wavefunction) [28–31]. Different locking strategies are possible with OpenMP. The most basic strategy is to use the `omp critical` construct, which in this example would synchronize the entire sigma vector. This is obviously a rather inefficient technique, given the typical size of said vector of several million elements. The odds of two

threads to hit the exact same element at the same time therefore seem rather slim. More fine-grained locking is possible using the `omp atomic` clause, which locks only a single memory address. Such fine-grained locking comes at the price of a high overhead of locks being manipulated for nothing. In our micro tests we found another locking strategy to be vastly superior: locking by blocks. To accomplish this, the vector is logically divided into different blocks of identical size, each with its own lock. Assessing the impact of different locking stride sizes will be one of the targets of the benchmarks presented in the results section.

Another important bottleneck to deal with besides locking is I/O, which is often forced to be a serial process. As the SGGA is formulated not in terms of the MO-transformed Cholesky vectors but the reassembled integrals, we already established in Ref. [20] the importance of an efficient buffering strategy even for the serial case. Although modern file systems buffer frequently accessed data in memory, a specialized buffering was found to be vastly superior. Our buffering strategy can be summarized as follows: instead of creating a normal Fortran random access file on disk, we interface with an I/O buffer library [20] that handles all required storage. We initialize a memory pool, providing used chunk size and a maximum amount of main memory to be used. A chunk of consecutive data typically contains hundreds of individual integral matrix elements. Upon opening a specific file, it is associated with a memory pool. Any subsequent read and write operations on that file are translated into an access of the appropriate memory chunk. The library API is chosen such that it strongly resembles traditional Fortran random disk file access. Storage "files" are identified with integer numbers, and data items are accessed by providing an index. Internally, the library keeps data chunks either in main memory or (as soon as the main memory provided to the pool is exhausted) on hard disk. Upon an access request, the appropriate chunk is fetched from the hard disk if necessary, and can then be processed. The typical access pattern we observe requires multiple accesses to several close-lying indices, before moving to a different file position. Our buffered scheme avoids multiple hard drive accesses in such cases. Per default, we use a last-in-first-out (LIFO) policy when deciding which chunk to free upon exhaustion of allowed main memory. While a last-in-last-out (LILO) policy might seem more appropriate, consecutive access of all vector elements (as typical in our use case) will cause hard-drive readings for all accesses in LILO, while the fraction of elements fitting in main memory at the beginning of the vector will be completely buffered using LIFO. Frequently accessed data are put into separate memory chunks, to ensure they are always fully in memory. Different files may be associated with different main memory chunks and storage policies.

Since I/O is intrinsically a serial process, care must be taken to avoid race conditions under concurrent manipulation of the data. Our I/O buffer library ensures this by use of proper locking whenever a chunk needs to be fetched from disk or committed to disk. For chunks in memory, concurrent reading operations or concurrent writing to different indices are not problematic. Loops are parallelized in such a way that concurrent writing to the same index (i.e., memory location) does not occur. We therefore do not need to safeguard against concurrent writing to the same index. Care must only be taken to guarantee that concurrent write accesses to different blocks that need to be fetched from the hard disk will not result in race conditions. We can thus guarantee very fast read/write accesses for chunks in memory, without the (significant) performance demand of locks. To avoid unnecessary disk I/O upon exhaustion of main memory if two cores are working on two different positions in the file, each core may keep one chunk in main memory. Consequently, if one core needs to access a new part of the written data, while other cores still work on data currently in memory, no unnecessary disk I/O operations occur. To further reduce checking of locks, we provide an API for directly loading (storing) an array of values from (into) the file. In this case, the library has to correctly track which chunks are still referenced by a thread. Several benchmark tests of the parallel version, for different systems, load levels, buffer sizes, and a number of cores, confirm that the parallel I/O buffer library correctly handles concurrent access requests. Performance benchmarking shows little dependence on chunk size, provided chunks are not too small (below 1 kB) since the hard disk is then accessed too often. We use 80 kB chunks. Due to the simple interface, replacing I/O file access by buffered I/O using our library should be straightforward for other legacy Fortran codes. Since available main memory has increased considerably in the past decade (although it is still more expensive than hard disk space), legacy codes might want to take advantage of the much faster access times of RAM as opposed to hard-drives (ratio $\approx 1{:}10^6$). While these can be migrated by clever usage of disk arrays for some cases [35–38], our library allows codes with moderate memory demands to be quickly adapted to keep all files entirely in memory, greatly increasing performance with minimal changes to the program. For more demanding problems, execution time can still be substantially improved by keeping often accessed memory regions in main memory.

## 3. Results and discussion

### 3.1. Performance assessment

This section assesses the performance of the parallelized subroutines and total times. As a well-controllable benchmark set, we use local singles and doubles configuration interaction (LSDCI) energy calculations of 1-alkynes with the full set of references (nine) obtained from a CAS[4e,4o]SCF reference wavefunction with a cc-pVDZ basis set. The alkynes are in their equilibrium structures, optimized at the UFF level of theory using Atomdroid [39,40]. We consider these to be good examples in size and difficulty for systems of current interest in, e.g., biodiesel combustion research.

All timings in these sections were obtained from averaging three independent runs, with thread-pinning enabled (scattering mode). Both the necessary CASSCF and Cholesky decomposition are carried out by the MOLCAS program package (version 7.8) [41] and are not accounted for in the total TigerCI wall times. The computational resources used were nodes with two Intel Xeon E5-2670 @ 2.60 GHz processors and up to 64 GB RAM. The integral buffer was allowed to occupy at most 20 GB of RAM. The optimized executables were compiled with ifort/icpc version 13.0 and linked against Intel MKL 11.0.

As discussed in the implementation details, the risk of race conditions arising from different threads concurrently manipulating the sigma vector is eliminated with a blockwise locking strategy. Consequently, performance might depend on the size of the blocks: on one hand, too small blocks will increase the overhead of the parallel code paths as they will require more lock manipulations. On the other hand, too large blocks will cause threads to block longer and hence cause a reduced parallel speedup due to unnecessary waiting cycles. We thus have to check how big both of these effects actually are, to judge whether coding an autotuning strategy to find an optimal block size would be justified. To do so, we benchmark the parallel speedup for different block sizes (see Figs. 1–3), and find limited impact of the locking stride size on the wall time of the total TigerCI execution time. In general, runtime deviations from the runtime at a stride size of 1024 elements lie within less than 5% for 1-decyne, 1-pentadecyne, and 1-icosyne. Larger deviations only occur for very small stride sizes of 16 or 32 elements that cause a substantial locking overhead. In general, the deviation data set is fairly rough and does not follow a clear pattern. We thus

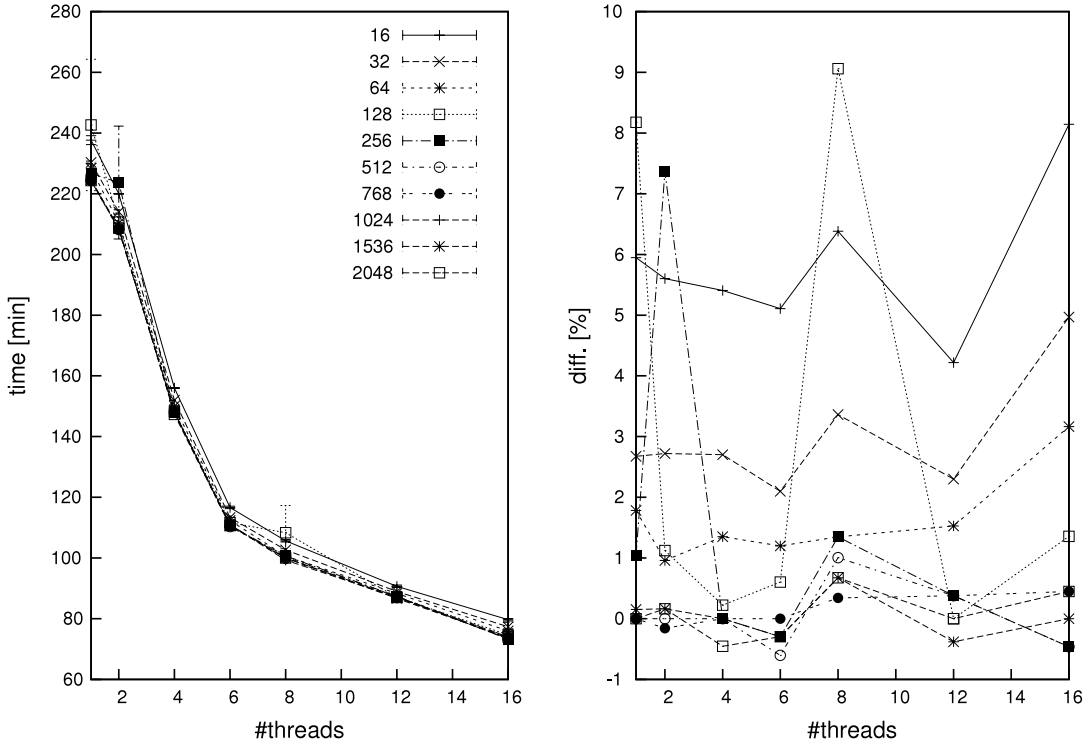



**Fig. 1.** Influence of the locking stride size for 1-decyne. Left panel: total wall times. Right panel: percent differences compared to the wall time obtained with a locking stride of 1024.

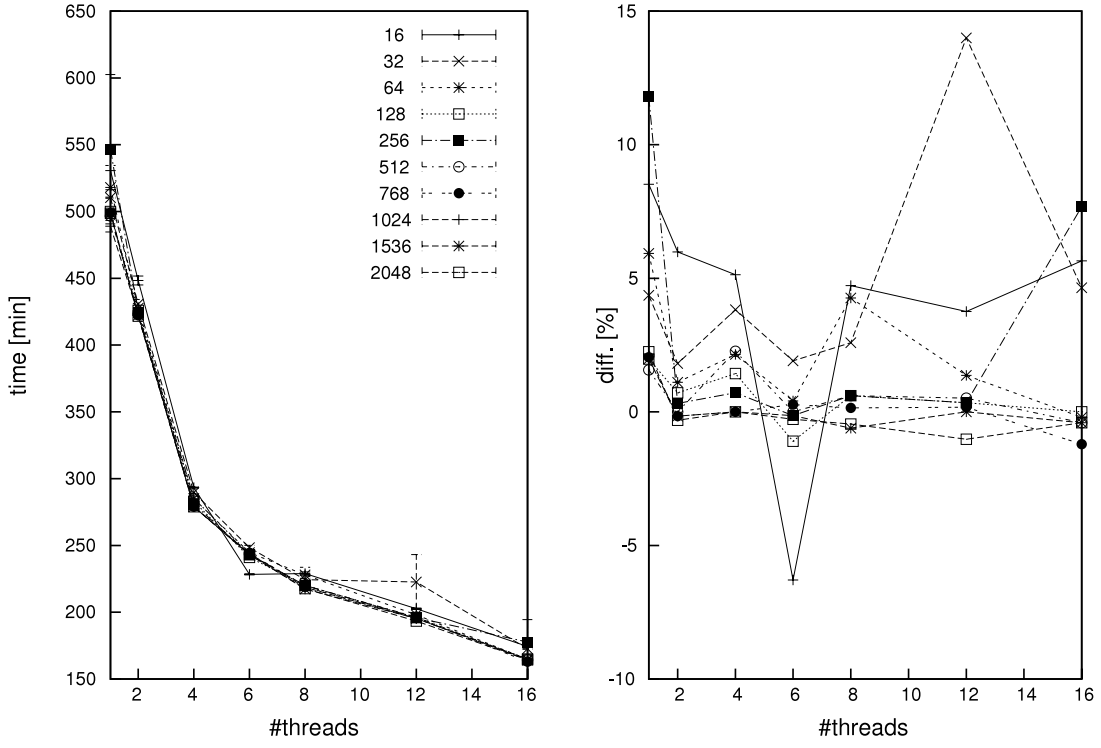



**Fig. 2.** Influence of the locking stride size for 1-pentadecyne. Left panel: total wall times. Right panel: percent differences compared to the wall time obtained with a locking stride of 1024.

expect a block size of 1024 elements to work sufficiently well for the systems under study here, without causing a substantial performance and/or efficiency penalty. We note here that in theory the optimal stride size should scale with the system size and should be larger in canonical (nonlocal) calculations than in local ones.

With the locking stride size set to 1024, an individual analysis of the parallelized parts of TigerCI can be carried out. This analysis will feature the wall time spent in each subroutine as a measure of its relative importance and the speedup achieved through shared-memory parallelization. Additionally, the CPU time spent in the

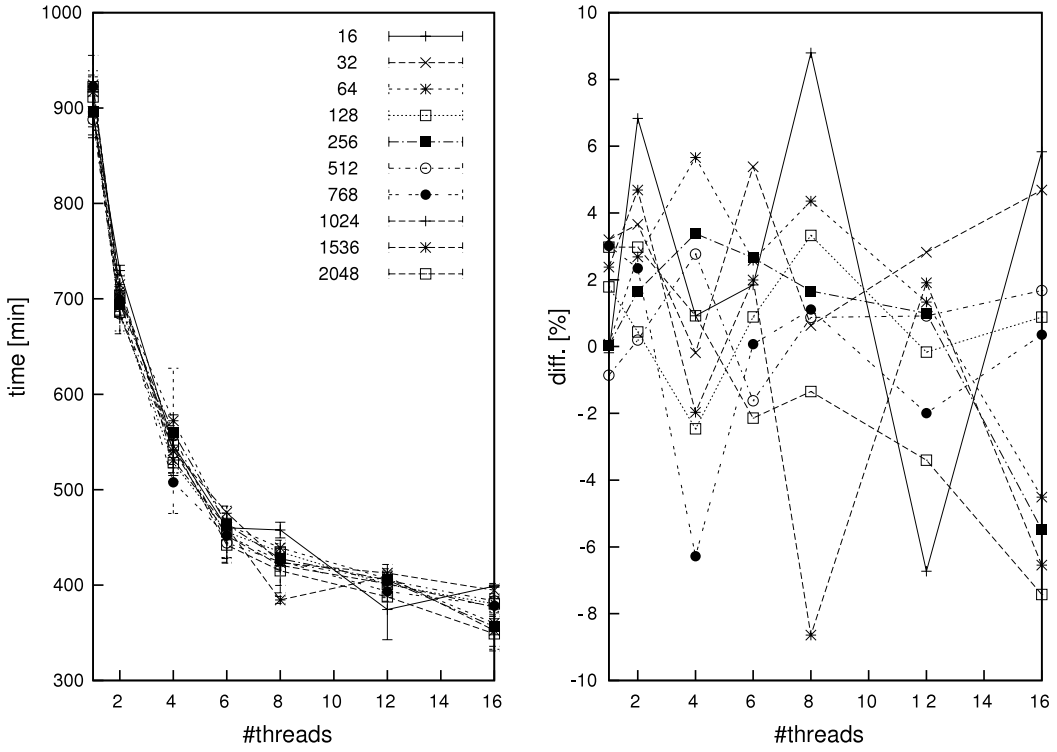



**Fig. 3.** Influence of the locking stride size for 1-icosyne. Left panel: total wall times. Right panel: percent differences compared to the wall time obtained with a locking stride of 1024.

routines is recorded and analyzed in terms of parallel overhead. Ideally, the CPU time spent in a kernel should stay constant, independent of the number of threads used to compute the kernel. An increase of the CPU time with the number of threads is indicative of parallelization-induced overhead, most notably, busy waits on locks and redundant computations. Obviously, only the most trivial parallel problems will exhibit an absolutely constant CPU time with the number of threads, while most real-world problems will show some increase. The magnitude of this increase is a very good measure to analyze and explain the source of suboptimal parallel speedups.

The AO to MO basis transform of the Cholesky vectors is the first major computational workload in TigerCI. This statement assumes that the AO Cholesky vectors were precalculated by MOLCAS when constructing the reference wavefunction. The transformation occurs in two steps by transforming the two AO indices individually:

$$T^I_{av} = \sum_{\mu} L^I_{\mu\nu} C_{\mu a}, \tag{1}$$

$$T^I_{ab} = \sum_{\nu} T^I_{a\nu} C_{\nu b}, \tag{2}$$

where $\mu$ and $\nu$ are AO indices, $a$ and $b$ are MO indices, and $I$ indexes the Cholesky vectors. The fully AO Cholesky vectors, $L^I_{\mu\nu}$, are transformed into the half AO, half MO order three tensor $T^I_{a\nu}$ in the first step. In the second step, $T^I_{a\nu}$ is transformed into the full MO Cholesky vector $T^I_{ab}$. To avoid storing the entire intermediate tensor $T^I_{a\nu}$, Eqs. (1) and (2) are computed in batches; each batch contains a subset of $a$'s. The amount of available memory controls the size of the transformation batches. In our tests, we allow a maximum of 10 GB to be used, allowing the transformation to be done in a single batch using the cc-pVDZ basis set. Both steps of the transformation are parallelized: Eq. (1) over the Cholesky index $I$ and Eq. (2) over the MO index $a$. A detailed analysis (Fig. 4) shows a favorable strong scaling (i.e., speedup by an increase in processor count at fixed problem size) already for the 1-decyne case with an 8-fold speedup for 16 threads. Unsurprisingly, the weak scaling (i.e., speedup by an increase in problem size at fixed processor count) is also favorable in this case, as the number of orbitals in the batch and the computational complexity of the kernel increases from 1-decyne to 1-icosyne. It is therefore advisable to maximize the number of orbitals treated in one transformation batch. The CPU time overhead is small and occurs due to the I/O of reading the AO Cholesky vectors from disk and writing the MO Cholesky vectors to the I/O buffer.

Since the SGGA is not formulated in terms of Cholesky vectors but in terms of actual integrals, the transformed Cholesky vectors need to be reassembled into the two-electron integrals. As highlighted in Ref. [20], this kernel is currently complicated by a sphere-based integral truncation employed by default which projects out certain integrals considered unimportant. Additionally, in preparation for the subsequent SGGA, the integrals are computed in classes. Each class covers one part of the total integral tensor. For example, one integral class consists of integrals with all orbital indices in the external space $(ab|cd)$ and another one of integrals with one index in the internal space and three indices in the external space, $(ia|bc)$. Therefore, the parallelization is carried out in two ways: (i) Integral classes covering a large part of the integral tensors are parallelized internally over the leading orbital index. This parallelization style is used for the three-external and four-external integrals. (ii) The other integral classes are evaluated in parallel, i.e., the parallelization is carried out on the integral class level. Fig. 5 provides an overview of the resulting parallel speedup. With a maximum speedup of more than 5 for 1-icosyne in our tests, this parallelization proves to be rather efficient given the inherent I/O bottleneck of the kernels. Note that our parallel I/O buffer is of crucial importance to achieve this speedup as it helps ease the serial nature of disk-based I/O.

Almost all the remaining computing time is spent in the SGGA. The SGGA implementation consists of thousands of lines of code and hundreds of subroutines. As discussed previously, a hotspot

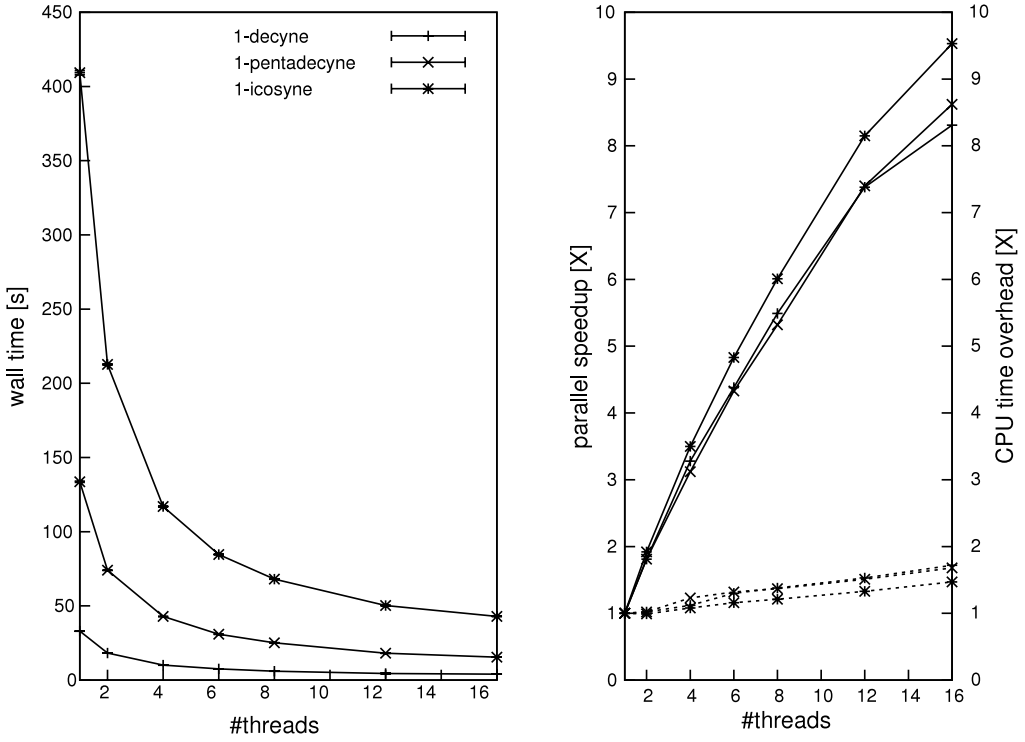



**Fig. 4.** Wall times, parallel speedup, and CPU time overhead for the transformation of Cholesky vectors from the AO into the MO basis. Left panel: wall times, right panel: parallel speedup and CPU overhead with solid lines used for speedup and dashed lines for overhead.

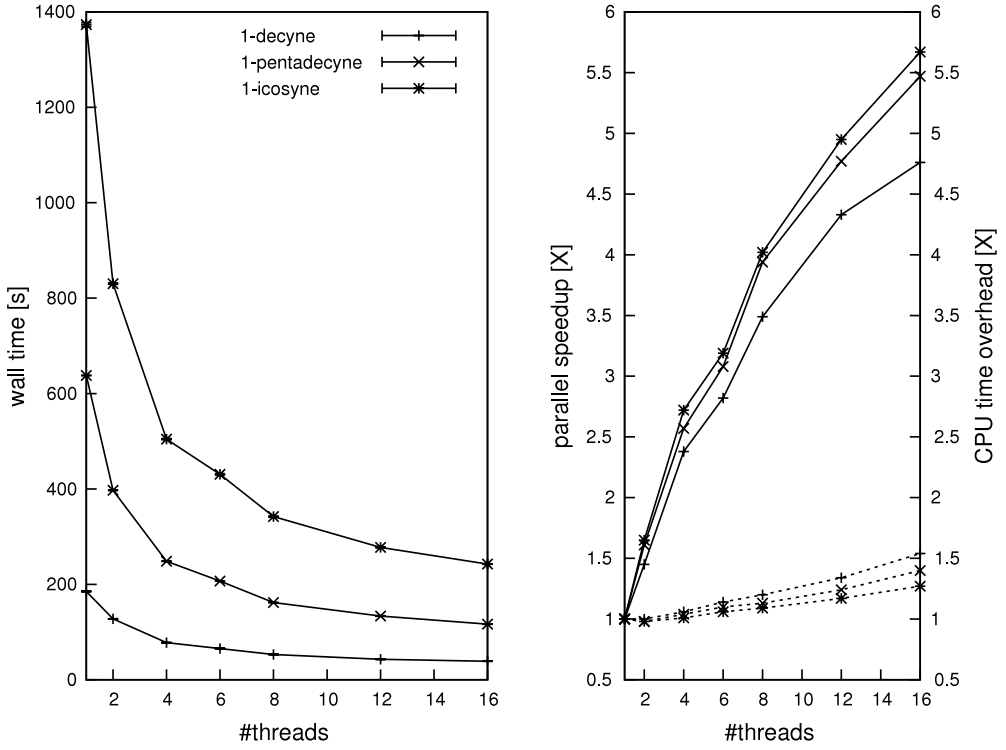



**Fig. 5.** Wall times, parallel speedup, and CPU time overhead for the integral reassembly from transformed Cholesky vectors. Left panel: wall times, right panel: parallel speedup and CPU overhead with solid lines used for speedup and dashed lines for overhead.

analysis was carried out to identify the subroutines where parallelization would benefit the total wall time. In the following, the parallel speedup achieved for these subroutines will be discussed in order of decreasing number of external orbital indices of the integrals used within the kernels. This order also corresponds (with one exception) to a decreasing time spent within that kernel. We refer to Ref. [20] for a detailed discussion of said analysis and the scaling behavior of different parts of the entire LSDCI algorithm with system size and number of references in the calculation.

Just as with the MO integral reassembly, the most expensive subroutine within the SGGA is typically the one dealing with four-external integral indices. This kernel is formulated vectorized [21]

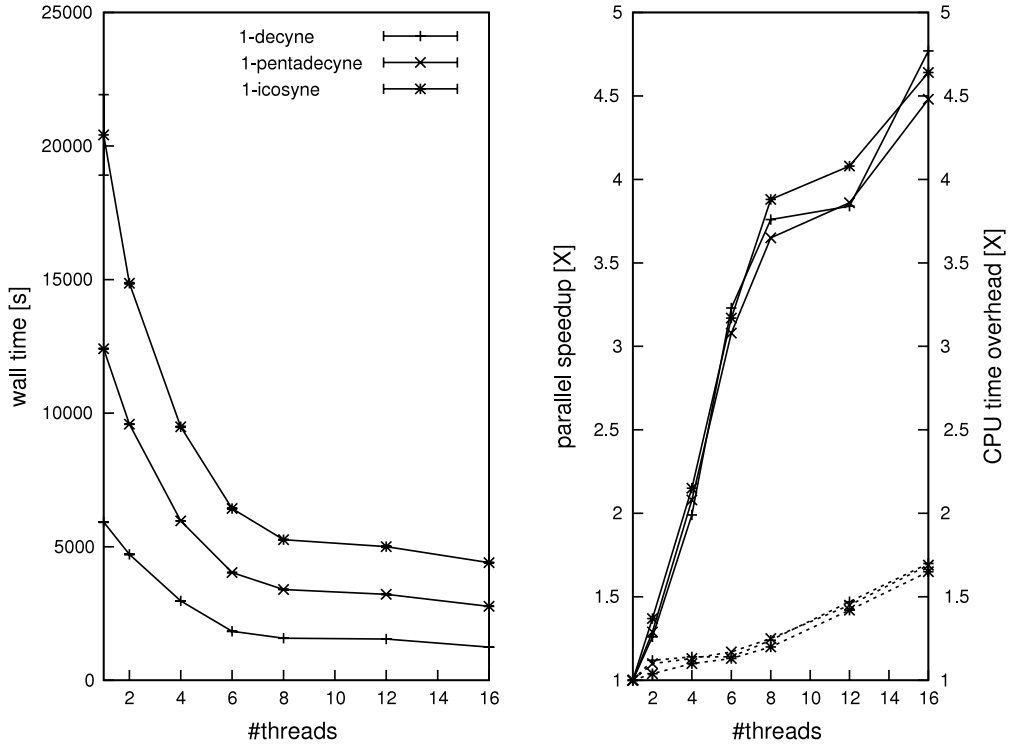



**Fig. 6.** Wall times, parallel speedup, and CPU time overhead for the four-external part of the SGGA. Left panel: wall times, right panel: parallel speedup and CPU overhead with solid lines used for speedup and dashed lines for overhead.

and integral-driven. Hence, the parallelization is over the leading external orbital index. This style of parallelization causes the reading of necessary integrals to lie within the parallel region. With our I/O buffer, such requests may be satisfied from fast main memory and in a concurrent fashion. The parallel efficiency of this kernel is at maximum 4.75 in our tests (see Fig. 6), with no notable beneficial weak scaling effects when enlarging the system from 1-decyne to 1-icosyne. The CPU time overhead is fairly limited, at a maximum of 1.75 in our tests.

The three-external kernel of the SGGA is again formulated in vectorized and integral-driven mode. The parallelization is over the leading internal orbital index. Again, the reading of necessary integrals is done within the parallel region from the I/O buffer. The parallel speedup of at most 2.5 is less than ideal for this routine in our tests (see Fig. 7). The CPU overhead is one of the reasons for this behavior, with a more than 2.5-fold CPU overhead for 16 threads in the 1-pentadecyne case. Although the locking of the sigma vector is done in an efficient way, it nevertheless penalizes the parallel throughput. This is also visible in the erratic wall time pattern, probably indicating blocking locks. Although this kernel is significantly less important than the four-external one in terms of computational cost, it nevertheless harms the total parallel speedup. It may be possible to obtain a more lock-free situation either through non-uniform stride sizes or reordering of the sigma vector.

The next vectorized kernel operates on two-internal/two-external integrals $(ij|ab)$. With a maximum serial wall time of 9000 s spent in this kernel compared to 12000 s spent in the three-external one, it is important for the overall parallel speedup. The parallelization is carried out over the leading internal index. We find a maximum speedup of 5 (see Fig. 8) with a beneficial weak scaling behavior for this kernel. By contrast to the three-external kernel, the CPU time overhead is again well-controlled at less than 1.25 in all our tests.

The three-internal subroutine is the only path-driven (not integral-driven) SGGA kernel that was parallelized in this work. Although the parallelization is formally injected over the number of internal orbitals, this is already at an inner loop level. This parallelization style causes a significant overhead in terms of CPU time spent for the solution (see Fig. 9). Logically, the maximum parallel efficiency is also very limited at less than two. The speedup even peaks at four to six threads depending on system size. Notably, the weak scaling exhibited by this kernel is actually disadvantageous. For these reasons, we limit the maximum number of threads allowed to be used in this kernel to four by default, independent of the number of threads used for the entire program execution. Additionally, a reformulation of this kernel into a more parallelization-friendly structure may be necessary. Fortunately, this kernel accounts for the smallest fraction of all kernels discussed here, making its impact on the overall scaling less severe.

We note one additional kernel where the path-based formulation is unsuitable for parallelization: the four-internal subroutine. Parallelization efforts carried out were able to obtain a kernel only with extremely limited scaling that even with 16 threads was significantly slower than the optimized serial kernel. A reformulation of such kernels in terms of a vectorized treatment may prove necessary in the future for specific large systems.

Both the three-internal as well as the four-internal kernels were found to scale sub-cubically with the system size and sub-quadratically with the number of references [20]. Therefore, their impact seems to be limited in the present context.

Finally, we discuss the parallelization of another vectorized kernel, the purely internal two-segment loops. We parallelize over the leading internal orbital index; as these integrals are small in number and therefore size, they are kept in memory during program execution. Note that the wall time spent within this kernel is two to three times higher than for the previously discussed three-internal kernel. The parallelization exhibits at maximum a parallel speedup of 3.2 with a reasonable scaling of the CPU time overhead (see Fig. 10). The weak scaling proves to be beneficial, improving the maximal speedup from less than 2.5 for 1-decyne to 3.2 for 1-icosyne. We can once more conclude that vectorized formulations of the SGGA kernels lend themselves well to parallelization and cause only limited overhead as opposed to path-based kernels.

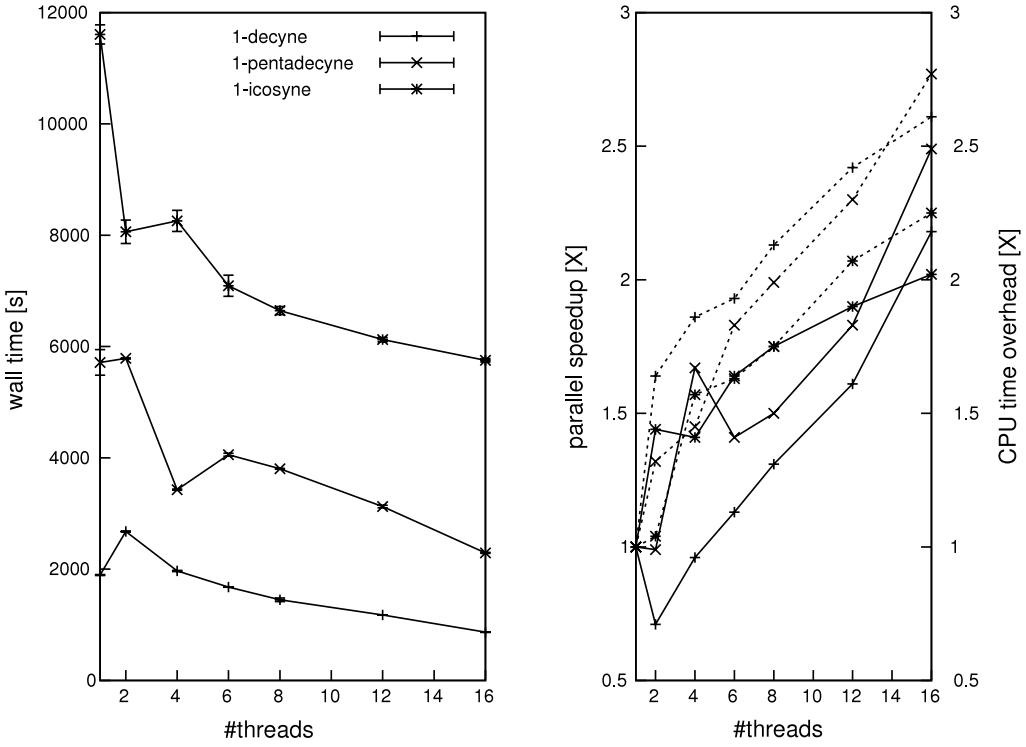



**Fig. 7.** Wall times, parallel speedup, and CPU time overhead for the three-external part of the SGGA. Left panel: wall times, right panel: parallel speedup and CPU overhead with solid lines used for speedup and dashed lines for overhead.

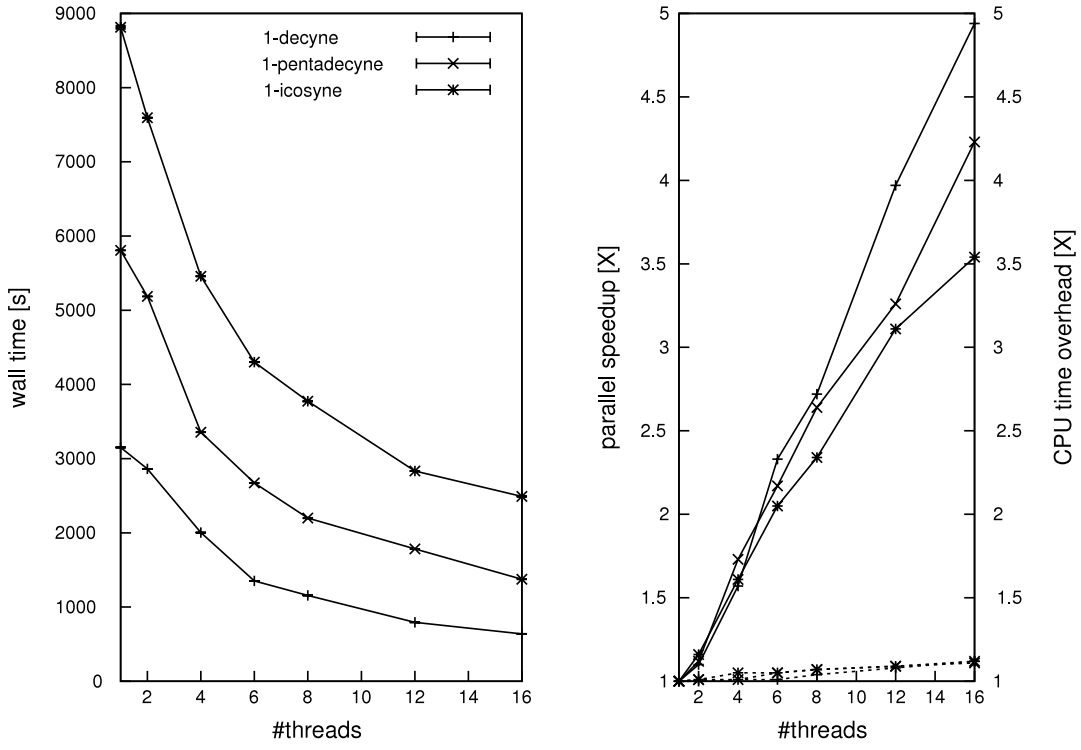



**Fig. 8.** Wall times, parallel speedup, and CPU time overhead for the two-internal part of the SGGA. Left panel: wall times, right panel: parallel speedup and CPU overhead with solid lines used for speedup and dashed lines for overhead.

Moving on to the overall picture, memory scaling with the number of threads is a very relevant yet seldom discussed feature of parallel implementations. Especially for algorithms requiring $\mathcal{O}(N^3)$ memory allocations for each thread, as the one discussed here, the memory requirement may become an obstacle for parallel program execution. An analysis of the scaling of the maximal resident set size (RSS) in Fig. 11 indicates that the added memory requirement for typical calculations is not prohibitive, with a maximum overhead of 25% for 16 threads in our setup. The maximum resident size is the maximal memory in main memory

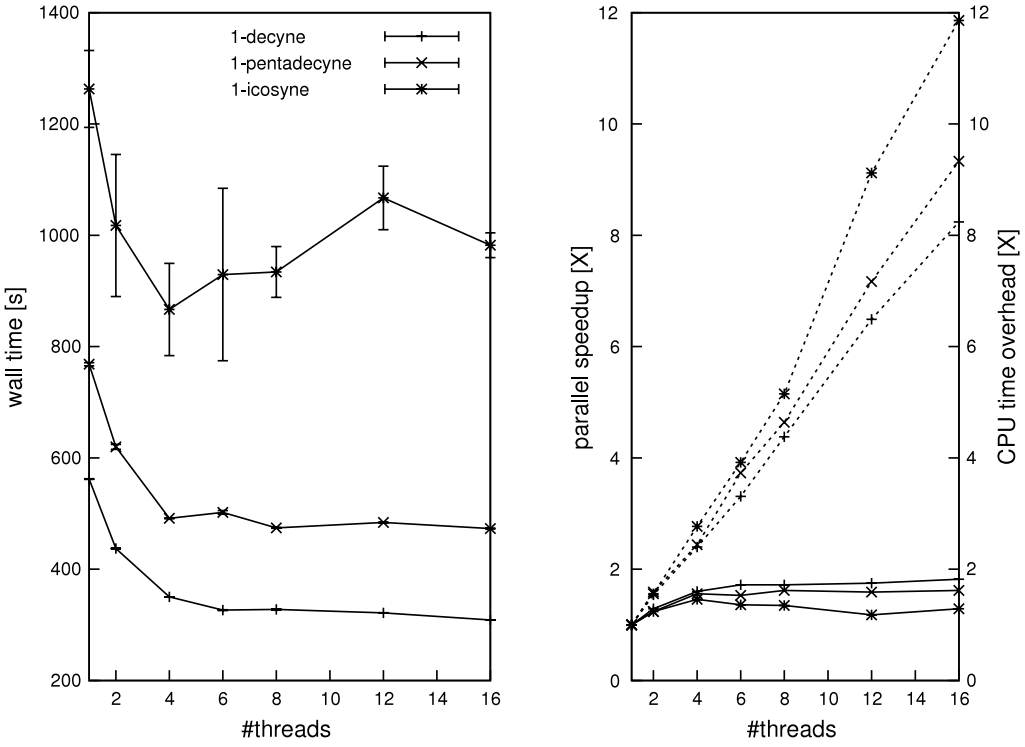



**Fig. 9.** Wall times, parallel speedup, and CPU time overhead for the three-internal part of the SGGA. Left panel: wall times, right panel: parallel speedup and CPU overhead with solid lines used for speedup and dashed lines for overhead.

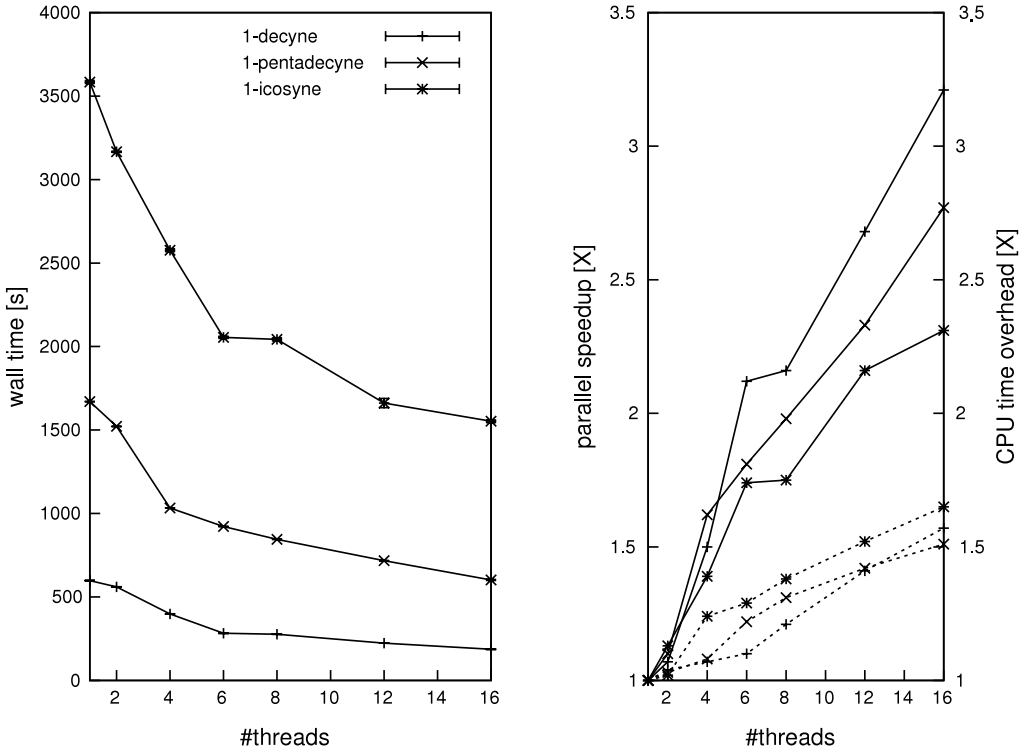



**Fig. 10.** Wall times, parallel speedup, and CPU time overhead for the purely internal part of the SGGA. Left panel: wall times, right panel: parallel speedup and CPU overhead with solid lines used for speedup and dashed lines for overhead.

associated with the process. After an offset phase, this overhead scales clearly linearly with the number of threads, which is the expected behavior.

We conclude the assessment of our parallel local CI implementation with overall scaling results. The total parallel speedup is mediocre, with a maximal speedup of three (see Figs. 12–14). As discussed previously, this result is due to the original SGGA implementation being heavily I/O bound and optimized for serial execution. Nevertheless, the obtained speedup is significant in practice. No significant difference between the test systems or beneficial weak scaling can be observed. The latter is in part caused by truncations initiated by the local algorithm. The structure of the

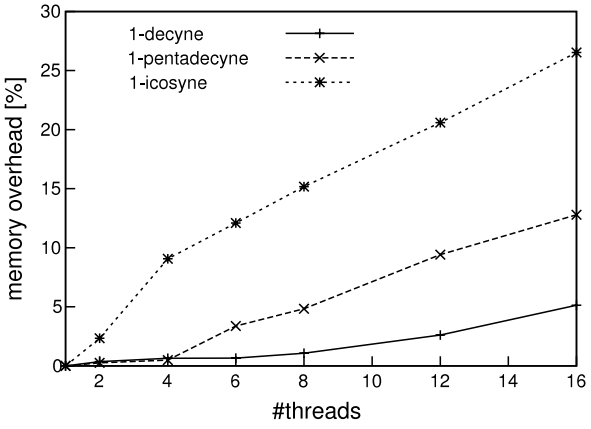



**Fig. 11.** Memory consumption as a function of the number of threads. Measured quantity is maximal resident set size during program execution, i.e., the maximal memory in main memory associated with the process. Single thread consumption for 1-decyne is 10.5 GB, for 1-pentadecyne 19.4 GB, and for 1-icosyne 29.9 GB.

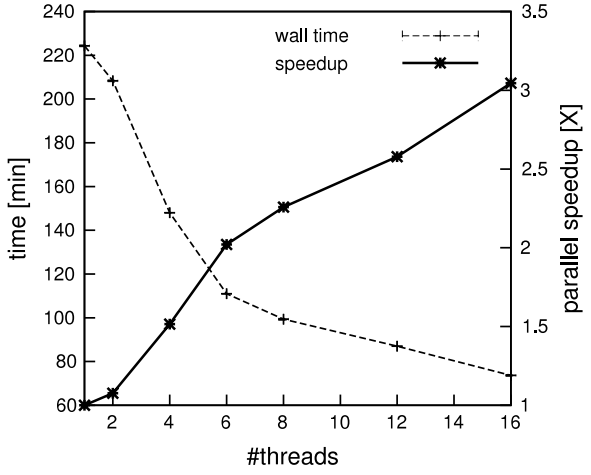



**Fig. 12.** Overall parallel speedup of the shared-memory parallelized TigerCI codebase for the 1-decyne test case.

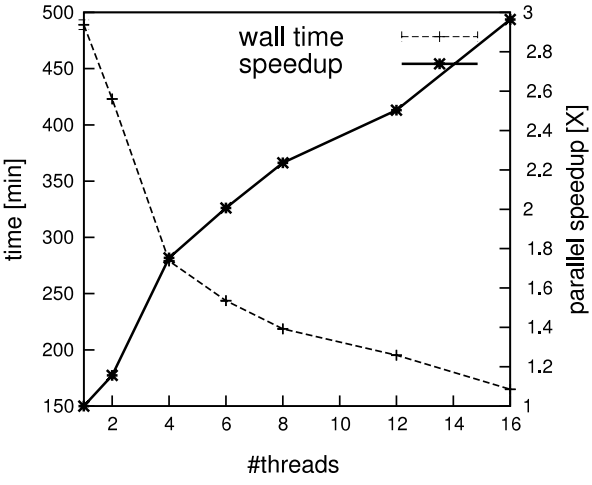



**Fig. 13.** Overall parallel speedup of the shared-memory parallelized TigerCI codebase for the 1-pentadecyne test case.

vectorized/integral-driven subroutines will cause maximal parallel speedup if the ratio of allowed excitations per orbital to the number of orbitals is high.

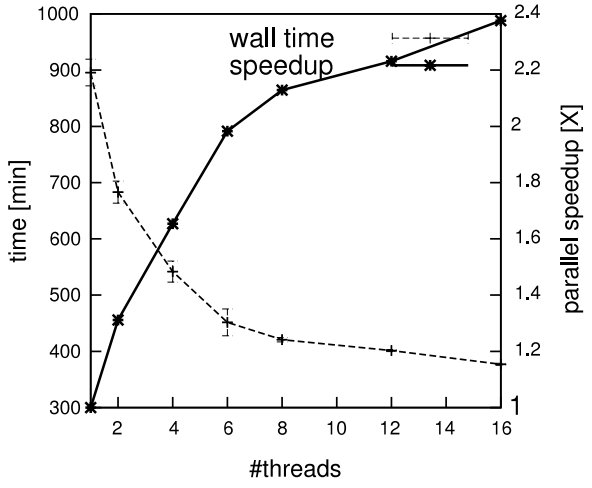



**Fig. 14.** Overall parallel speedup of the shared-memory parallelized TigerCI codebase for the 1-icosyne test case.

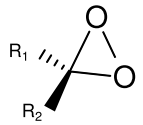



**Fig. 15.** General structure of dioxirane. $R_1$ and $R_2$ are hydrogen atoms or organic substituents.

### 3.2. Applications of parallel, local MRSDCI theory

#### 3.2.1. Localized MRSDCI Study of Alkyl(trifluoromethyl)dioxirane and its reaction with chloride ion

Dioxiranes (see Fig. 15) are reactive, yet mild and selective oxidants of organic compounds. In green oxidation technologies for water treatment, dioxirane derivatives are emerging as fast-acting, chlorine-free, and environmentally-friendly oxidizing agents for disinfection in aqueous media effective for destruction of various strains of microorganisms [42]. The characteristic reactions of dioxiranes [43] are: electrophilic transfer of an oxygen atom to nucleophilic substrates (e.g., epoxidation of alkenes) (Fig. 16, reaction (a)); insertion of an oxygen atom into C–H or Si–H sigma bonds in alkanes and silanes (Fig. 16(b) and (c)); and oxidation of chloride ions to hypochlorite ions (Fig. 16(d)). The reactivity and selectivity of dioxiranes can be fine-tuned by choosing suitable substituents; for example, methyl(trifluoro-methyl)dioxirane ($R_1 = CH_3$, $R_2 = CF_3$ in Fig. 15) exhibits higher reactivity than dimethyldioxirane [44], while dioxiranes with a *tert*-butyl substituent ($R_1 = (CH_3)_3C$ in Fig. 15) show higher selectivity than dimethyldioxirane [45]. For industrial applications such as cleaning, disinfection, and decontamination, alkyl(trifluoromethyl)dioxirane oxidants ($R_1$ = alkyl, $R_2 = CF_3$ in Fig. 15) were suggested as safer, more selective, and more effective alternatives to conventional hypochlorite- and peroxide-based agents [46].

The molecular structure of dioxiranes is characterized by a strained O–O bond having noticeable diradical character [47], thus necessitating the use of multireference methods for computational studies of these compounds. The local MRCI method is a natural choice for the study of alkyl-substituted dioxiranes. In this paper, the local MRSDCI (LMRSDCI) method is employed to compute the energy of the oxidation reaction, in which *n*-butyl(trifluoromethyl)dioxirane (**I**) oxidizes the chloride ion to form hypochlorite ion and *n*-butyl(trifluoromethyl)ketone (**II**) as

**Table 1**
CASSCF, LMRSDCI, and dynamic correlation energies (all in hartrees) for compound **I** and compound **II** and the reaction energies computed using Eq. (4), with the value of $(E_{OCl^-} - E_{Cl^-})$ set to −75.0253790 a.u.

| | Compound **I** | Compound **II** | Reaction energy (kcal/mol) |
|---|---|---|---|
| Reference CASSCF energy | −680.444201 | −605.654703 | **−148.0** |
| Weight of RHF reference in CASSCF | 0.58 | 1.0 | |
| LMRSDCI energy | −681.751107 | −606.866757 | **−88.5** |
| Weight of the reference in LMRSDCI | 0.79 | 0.80 | |
| Dynamic correlation energy | −1.306906 | −1.212054 | |
| Wall-clock time (min) | 40.0 | 5.3 | |

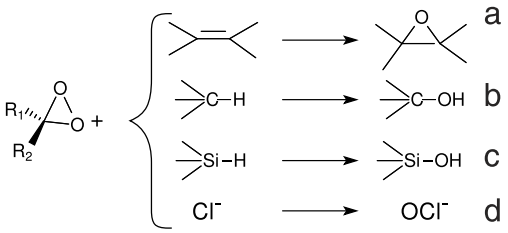
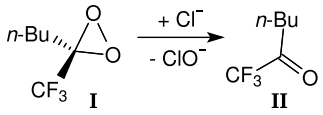



**Fig. 16.** Characteristic reactions of dioxiranes: (a) epoxidation of alkenes; (b) insertion of an oxygen atom into a C–H sigma bond; (c) insertion of an oxygen atom into a Si–H sigma bond; (d) oxidation of chloride ion to the hypochlorite ion.

products (Eq. (3)).




(3)

The molecular geometries of *n*-butyl(trifluoromethyl)-dioxirane (**I**) and *n*-butyl(trifluoromethyl) ketone (**II**) were optimized using density functional theory (DFT) with the B3LYP functional [48–50] and the 6-31G atomic basis set [51,52]; the optimization was conducted using the GAMESS [53,54] quantum chemistry program. The optimized structures are shown in Fig. 17.

Considering the bonding and anti-bonding sigma orbitals of the O–O bond as active orbitals (see Fig. 18), a complete active space self-consistent field (CASSCF) computation in a CAS[2e,2o] active space was carried out using the cc-pVDZ basis set [55] to generate the reference multiconfigurational wavefunction, followed by an LMRSDCI calculation.

CASSCF reference energies, LMRSDCI energies, dynamic correlation energies, and an estimated reaction (Eq. (3)) energy are presented in Table 1. The dynamic correlation energy is defined as the difference between the LMRSDCI and the energies of the reference wavefunctions. The reaction energy is estimated as the difference between the total energies of the reactants and products, corrected by the energy difference between chloride and hypochlorite ions:

$$E_{\text{reaction}} = E_{\text{II}} - E_{\text{I}} + (E_{OCl^-} - E_{Cl^-}). \quad (4)$$

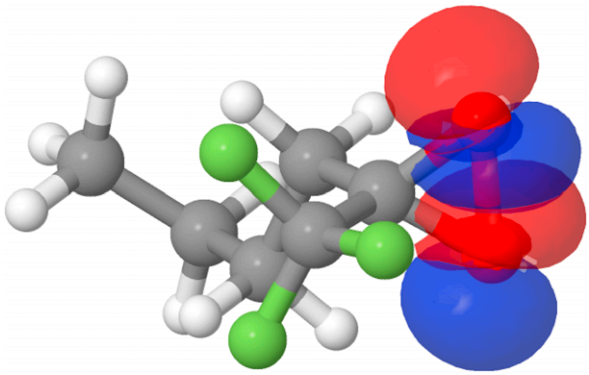


**Fig. 18.** Localized active orbitals of *n*-butyl(trifluoromethyl)dioxirane. Carbon, hydrogen, oxygen, and fluorine atoms are represented as gray, white, red, and green balls respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)
*Source:* The picture is generated by Jmol [66].

$E_{\text{reaction}}$ is the energy of the reaction; $E_{\text{I}}$ and $E_{\text{II}}$ are the total energies of the reactant and the product, respectively; $E_{OCl^-}$ and $E_{Cl^-}$ are the energies of chloride and hypochlorite ions. The CCSD(T)/aug-cc-pVTZ energies of $E_{OCl^-} = -534.830449$ and $E_{Cl^-} = -459.805070$ ha are reported in the NIST Computational Chemistry Comparison and Benchmark Database (CCCBDB) [56].

The CASSCF calculation shows that compound **I** indeed exhibits multireference character with a reference weight of the restricted Hartree–Fock (RHF) determinant to be around 58%. The reaction energy computed at the CASSCF level is −148 kcal/mol, suggesting that the reaction is thermodynamically favorable. The LMRSDCI method predicts the reaction energy at the correlated level to be −88.50 kcal/mol, showing a substantial dynamic correlation effect (59.5 kcal/mol) on the reaction energy; this is also corroborated by a noticeable (20%) weight of out-of-reference singly and doubly excited configurations. On the other hand, the reasonably high weight of the reference wavefunction in the LMRSDCI wavefunction indicates the adequacy of the chosen active space. A short wall clock time of 40 min, using eight threads on a dual Intel Xeon E5450 system with 16 GB RAM, for a multireference

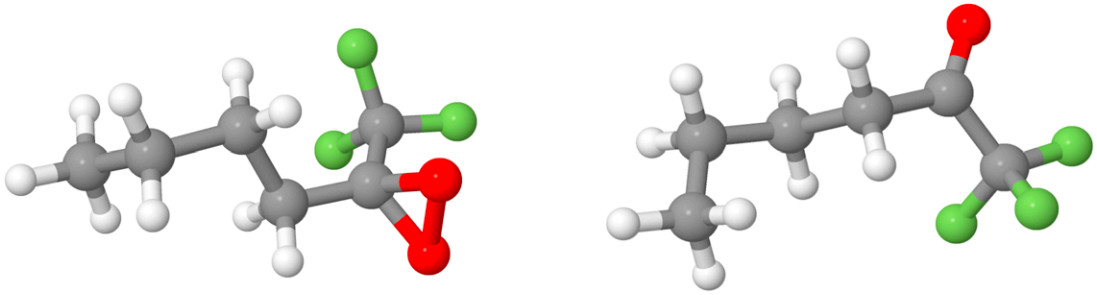


**Fig. 17.** The optimized structures of *n*-butyl(trifluoromethyl)dioxirane (left) and *n*-butyl(trifluoromethyl)ketone (right). Carbon, hydrogen, oxygen, and fluorine atoms are represented as gray, white, red, and green balls, respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)
*Source:* Pictures are generated by Jmol [66].

**Table 2**
CASSCF, LMRSDCI, and dynamic correlation energies (all in hartrees) for solvated and unsolvated methyl nitrene, and the calculated values of the singlet–triplet gap.

| | CASSCF energy | LMRSDCI energy | Dynamic correlation energy | CASSCF reference weight | Time (min) |
|---|---|---|---|---|---|
| Unsolvated $CH_3N$ | | | | | |
| $T_0$ | −94.001474 | −94.266483 | −0.265010 | 0.91 | 0.15 |
| $S_1$ | −93.936723 | −94.208585 | −0.271862 | 0.91 | 0.10 |
| $S_2$ | −93.874506 | −94.164885 | −0.290379 | 0.90 | 0.25 |
| S–T gap (kcal/mol) | **40.6** | **36.3** | | | |
| $(H_2O)_{10}CH_3N$ | | | | | |
| $T_0$ | −854.323975 | −856.467726 | −2.143751 | 0.70 | 5 |
| $S_1$ | −854.259967 | −856.402060 | −2.142093 | 0.69 | 10 |
| $S_2$ | −854.198945 | −856.385139 | −2.186194 | 0.69 | 9 |
| S–T gap (kcal/mol) | **40.2** | **41.2** | | | |
| $(H_2O)_{15}CH_3N$ | | | | | |
| $T_0$ | −1234.502081 | −1237.350888 | −2.848807 | 0.82 | 18 |
| $S_1$ | −1234.438135 | −1237.312943 | −2.874808 | 0.67 | 30 |
| $S_2$ | −1234.377183 | −1237.261462 | −2.884279 | 0.67 | 31 |
| S–T gap (kcal/mol) | **40.1** | **23.8** | | | |

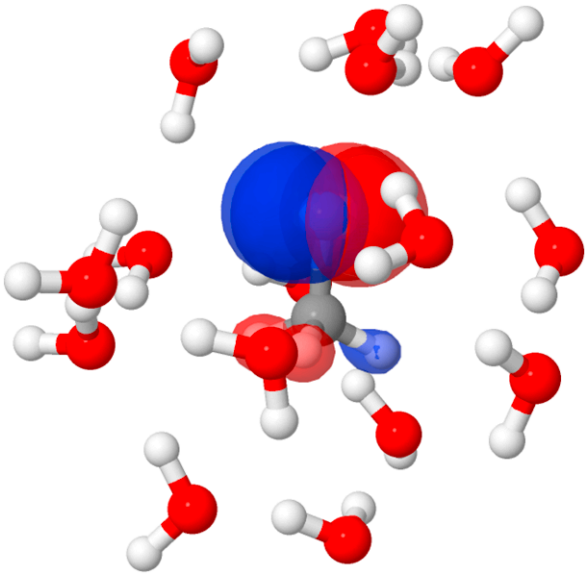


**Fig. 19.** Localized active orbitals of $CH_3N(H_2O)_{15}$. Carbon, hydrogen, oxygen, and nitrogen atoms are represented as gray, white, red, and blue balls respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)
*Source:* The picture is generated by Jmol [66].

computation suggests that the computation of long chain alkyl substituted dioxiranes is feasible using the LMRSDCI method.

### 3.2.2. LMRSDCI study of singlet–triplet splitting energy gap of methyl nitrene, methyl nitrene–$(H_2O)_{10}$, and methyl nitrene–$(H_2O)_{15}$ clusters

Nitrenes are reactive diradical intermediates of general formula R–N (where R is a hydrogen atom or an organic radical) that find uses in synthesis, polymer cross-linking, photoaffinity labeling, and photoresist technology [57–59]. Nitrenes may exist in either a singlet or triplet electronic state: singlet alkyl nitrenes are formed as a result of photodecomposition of the corresponding azines, and lower-energy triplet alkyl azines were identified both in the gas phase and in solution. It is expected that triplet alkyl nitrenes are longer-lived than their singlet forms and are more easily intercepted by chemical reactions [60]. Due to their diradical nature, computational investigations of the electronic structure of nitrenes require the use of multireference methods [61]. As the solvent environment can be expected to affect the electronic properties and reactivity of nitrenes [60], it is imperative to include solvent molecules in the computational model. In particular, the closest solvent molecules are required to be included in the *ab initio* treatment [62]. The LMRSDCI method was employed to compute the singlet–triplet energy gap of methylnitrene $CH_3N$ and aqueous methylnitrene clusters $CH_3N(H_2O)_{10}$ and $CH_3N(H_2O)_{15}$.

The molecular geometry of methyl nitrene for the lowest energy triplet state was optimized using unrestricted density functional theory (UDFT) with the B3LYP functional [48–50] and the 6-31G basis set [51,52] (UDFT/B3LYP/6-31G); the optimization was conducted using the GAMESS [53,54] quantum chemistry program.

To obtain the relaxed geometry of methyl nitrene in the presence of solvent, the quantum mechanics/effective fragment potential (QM/EFP1) method [63] was used to solvate the methyl nitrene. The UDFT/B3LYP/6-31G/EFP1 model containing the UDFT/B3LYP/6-31G QM methyl nitrene part and 50 EFP1 water molecules was constructed. An NVT-ensemble [64] QM/EFP1/molecular dynamics (MD) simulation was carried out for the model for 3 picoseconds using a Nosé–Hoover thermostat [64] at 300 K. Then, the structures of methyl nitrene with 15 and 10 closest water molecules, as well as the unsolvated methyl nitrene structure, were cut out from an MD snapshot for the CASSCF and LMRSDCI computations.

To generate the reference multiconfigurational wavefunction, CASSCF computations were carried out in a CAS[2e,2o] active space with the cc-pVDZ basis [55]. The two electrons are distributed in an active space consisting of 2 molecular orbitals with the maximum contributions of $p_x$ and $p_y$ nitrogen orbitals (see Fig. 19). Two state-specific CASSCF computations were performed to obtain two singlets (as expected, the lowest two CASSCF roots are nearly degenerate, corresponding to one approximately doubly degenerate singlet state $S_1$). A state-specific CASSCF computation was performed to obtain one triplet state. As expected, the ground state of methyl nitrene is a triplet state $T_0$.

For the three chosen structures of $CH_3N$, $CH_3N(H_2O)_{10}$, and $CH_3N(H_2O)_{15}$ clusters, the CASSCF reference energies, LMRSDCI energies, dynamical correlation energies for $T_0$, $S_1$, and $S_2$ states, and the singlet–triplet energy gap (S–T gap) are presented in Table 2. LMRSDCI predicts the S–T gap for the gas phase $CH_3N$ to be 36.3 kcal/mol, in reasonable agreement with the experimental value of 31.2(±0.3) kcal/mol [65]. The S–T gap increases for the $CH_3N(H_2O)_{10}$ cluster and then decreases for $CH_3N(H_2O)_{15}$, suggesting that the S–T gap of $CH_3N$ is substantially influenced by the degree of solvation (the number and positions of surrounding water molecules). At the reasonably solvated level of 15 waters, the $S_1$ state of the methyl nitrene molecule is stabilized (relative to the $T_0$ state) by the surrounding water molecules.

The relative stabilization of the $S_1$ state of the methyl nitrene molecule immersed in a polarizable medium, such as water, is consistent with the higher dipole moment of the singlet state

(2.07 D for the singlet state *vs.* 1.98 D for the triplet state, according to the CASSCF calculation). The predicted increase of the S–T gap in the case of the $CH_3N(H_2O)_{10}$ cluster can be attributed to a number of factors, such as incomplete solvation, the energetically unfavorable interaction of the methyl nitrene molecule with permanent dipoles of water molecules, and the absence of the polarizable medium in cluster models beyond the 10 or 15 surrounding water molecules. It is also reasonable to expect that geometry relaxation or a proper configurational sampling of the solvation shell will affect the calculated S–T gap. However, a systematic investigation of the influence of these factors is beyond the scope of the present work. The wall clock time of tens of minutes, using eight threads on a dual Intel Xeon E5450 system with 16 GB RAM, suggests that the LMRSDCI method is a promising approach for correlated computations with inclusion of explicit solvent molecules.

## 4. Conclusions

We have shown here a first implementation of a shared-memory local multi-reference CI extension to our TigerCI code. The speedups achieved in the parallelized routines are mediocre to good with the total speedup of typical benchmark calculations to be three-fold on our 16 core benchmarking system. We are able to attribute this unfavorable scaling to lock-contention in specific routines and the general design of the SGGA being intrinsically hard to efficiently parallelize. Although the parallel speedups reported here are not close to linear, we believe they will in practice, in conjunction with our reduced scaling CI implementation, enable science for significantly larger systems than are currently accessible though canonical multireference codes.

Within the context of a Cholesky-decomposed, SGGA-based local CI implementation, we expect improvements through reformulations of the routines posing bottlenecks in terms of a vectorized mode. Additionally, a reformulation of the numerical kernels in terms of the MO transformed Cholesky vectors could help to overcome the remaining I/O induced lock contention and in general improve execution time. In the meantime, an integral-direct implementation could be used to lift some of the restrictions currently present in the code.

## Acknowledgments

JMD acknowledges a German academic exchange service (DAAD) fellowship. EAC thanks the US National Science Foundation (Grant No. CHE-1265700) for support of this work. All calculations presented in the performance assessment section were carried out using Princeton's TIGRESS High Performance Computing resources.

MSG, TLW and AG were supported by a grant from the US Department of Energy, Office of Basic Energy Sciences, Division of Chemical Sciences, Geosciences and Biosciences through the Ames Laboratory PCTC, Chemical Physics, and Homogeneous and Interfacial Catalysis project. The Ames Laboratory is operated for the US Department of Energy by Iowa State University under contract No. DE-AC02-07CH11358. The calculations presented in the applications section were performed on a Linux cluster that was provided by a Department of Defense DURIP grant.

## References

[1] G.E. Moore, Proc. IEEE 86 (1998) 82.
[2] K.R. Shamasundar, G. Knizia, H.-J. Werner, J. Chem. Phys. 135 (2011) 054101. URL: http://scitation.aip.org/content/aip/journal/jcp/135/5/10.1063/1.3609809.
[3] J. Brabec, J. Pittner, H.J.J. van Dam, E. Aprà, K. Kowalski, J. Chem. Theory Comput. 8 (2012) 487. http://pubs.acs.org/doi/pdf/10.1021/ct200809m, URL: http://pubs.acs.org/doi/abs/10.1021/ct200809m.
[4] H. Dachsel, H. Lischka, R. Shepard, J. Nieplocha, R.J. Harrison, J. Comput. Chem. 18 (1997) 430.
[5] H. Lischka, T. Müller, P.G. Szalay, I. Shavitt, R.M. Pitzer, R. Shepard, Wiley Interdiscip. Rev.: Computat. Mol. Sci. (ISSN: 17590876) 1 (2011) 191. URL: http://doi.wiley.com/10.1002/wcms.25.
[6] P. Pulay, Chem. Phys. Lett. 100 (1983) 151. URL: http://www.sciencedirect.com/science/article/pii/0009261483807039.
[7] S. Saebo, P. Pulay, Annu. Rev. Phys. Chem. (ISSN: 0066-426X) 44 (1993) 213. URL: http://www.annualreviews.org/doi/abs/10.1146/annurev.pc.44.100193.001241.
[8] S. Saebø, P. Pulay, Chem. Phys. Lett. 113 (1985) 13. URL: http://www.sciencedirect.com/science/article/pii/000926148585003X.
[9] M. Schutz, G. Hetzer, H.-J. Werner, J. Chem. Phys. (ISSN: 00219606) 111 (1999) 5691. URL: http://link.aip.org/link/JCPSA6/v111/i13/p5691/s1&Agg=doi.
[10] M. Schutz, H.-J. Werner, J. Chem. Phys. (ISSN: 00219606) 114 (2001) 661. URL: http://link.aip.org/link/JCPSA6/v114/i2/p661/s1&Agg=doi.
[11] E.A. Carter, D. Walter, Encyclopedia Comput. Chem. (2004) URL: http://onlinelibrary.wiley.com/doi/10.1002/0470845015.cu0024/full.
[12] D.G. Liakos, A. Hansen, F. Neese, J. Chem. Theory Comput. 7 (2011) 76. http://pubs.acs.org/doi/pdf/10.1021/ct100445s, URL: http://pubs.acs.org/doi/abs/10.1021/ct100445s.
[13] A. Venkatnathan, A.B. Szilva, D. Walter, R.J. Gdanitz, E.A. Carter, J. Chem. Phys. (ISSN: 0021-9606) 120 (2004) 1693. URL: http://www.ncbi.nlm.nih.gov/pubmed/15268300.
[14] D. Walter, A.B. Szilva, K. Niedfeldt, E.A. Carter, J. Chem. Phys. (ISSN: 00219606) 117 (2002) 1982. URL: http://link.aip.org/link/JCPSA6/v117/i5/p1982/s1&Agg=doi.
[15] D. Walter, A. Venkatnathan, E.A. Carter, J. Chem. Phys. (ISSN: 00219606) 118 (2003) 8127. URL: http://link.aip.org/link/JCPSA6/v118/i18/p8127/s1&Agg=doi.
[16] T.S. Chwee, A.B. Szilva, R. Lindh, E.A. Carter, J. Chem. Phys. (ISSN: 1089-7690) 128 (2008) 224106. URL: http://www.ncbi.nlm.nih.gov/pubmed/18554005.
[17] T.S. Chwee, E.A. Carter, J. Chem. Phys. (ISSN: 1089-7690) 132 (2010) 074104. URL: http://www.ncbi.nlm.nih.gov/pubmed/20170212.
[18] T.S. Chwee, E.A. Carter, J. Chem. Theory Comput. (ISSN: 1549-9618) 7 (2011) 103. URL: http://pubs.acs.org/doi/abs/10.1021/ct100486q.
[19] D.B. Krisiloff, E.A. Carter, Phys. Chem. Chem. Phys. (ISSN: 1463-9084) 14 (2012) 7710. URL: http://xlink.rsc.org/?DOI=c2cp23757a. http://www.ncbi.nlm.nih.gov/pubmed/22358179.
[20] D.B. Krisiloff, J.M. Dieterich, F. Libisch, E.A. Carter, in: R. Melnick (Ed.), Mathematical and Computational Modeling, Wiley, 2014 (in press).
[21] D. Walter, E.A. Carter, Chem. Phys. Lett. (ISSN: 00092614) 346 (2001) 177. URL: http://linkinghub.elsevier.com/retrieve/pii/S0009261401009666.
[22] The openmp api specification for parallel programming, http://openmp.org/wp (2014).
[23] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI: The Complete Reference, MIT Press, Cambridge, MA, USA, 1995.
[24] D.G. Fedorov, R.M. Olson, K. Kitaura, M.S. Gordon, S. Koseki, J. Comput. Chem. (ISSN: 1096-987X) 25 (2004) 872. URL: http://dx.doi.org/10.1002/jcc.20018.
[25] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, E. Apra, Int. J. High Perform. Comput. Appl. 20 (2006) 203.
[26] M. Baker, M. Grove, A. Shafi, Parallel and Distributed Computing, 2006. ISPDC '06. The Fifth International Symposium on (2006), pp. 3–10.
[27] pympi: Putting the py in mpi, http://pympi.sourceforge.net/index.html (2014).
[28] W. Duch, J. Karwowski, Theor. Chem. Acc.: Theory, Comput. Model. (Theor. Chim. Acta) 51 (1979) 175. URL: http://www.springerlink.com/index/P72R78TN38X23Q72.pdf.
[29] W. Duch, J. Karwowski, Int. J. Quantum Chem. (ISSN: 0020-7608) 22 (1982) 783. URL: http://doi.wiley.com/10.1002/qua.560220411.
[30] W. Duch, J. Karwowski, Comput. Phys. Rep. (ISSN: 01677977) 2 (1985) 93. URL: http://linkinghub.elsevier.com/retrieve/pii/0167797785900012.
[31] W. Duch, J. Karwowski, Theor. Chim. Acta (ISSN: 0040-5744) 71 (1987) 187. URL: http://www.springerlink.com/index/10.1007/BF00526416.
[32] K. Tanaka, Y. Mochizuki, T. Ishikawa, H. Terashima, H. Tokiwa, Theor. Chem. Acc. 117 (3) (2006) 397–405. http://dx.doi.org/10.1007/s00214-006-0171-8.
[33] J.M. Juran, Quality Control Handbook, McGraw-Hill Book Company, New York, 1951.
[34] G.M. Amdahl, AFIPS Conf. Proc. 30 (1967) 483.
[35] A. Asadchev, M.S. Gordon, J. Chem. Theory Comput. 9 (2013) 3385. http://pubs.acs.org/doi/pdf/10.1021/ct400054m, URL: http://pubs.acs.org/doi/abs/10.1021/ct400054m.
[36] T. Janowski, A.R. Ford, P. Pulay, J. Chem. Theory Comput. 3 (2007) 1368.
[37] T. Janowski, P. Pulay, J. Chem. Theory Comput. 4 (2008) 1585.
[38] J. Baker, T. Janowski, K. Wolinski, P. Pulay, Comput. Mol. Sci. 2 (2012) 63.
[39] A.K. Rappe, C.J. Casewit, K.S. Colwell, W.A. Goddard, W.M. Skiff, J. Am. Chem. Soc. 114 (1992) 10024. http://pubs.acs.org/doi/pdf/10.1021/ja00051a040. URL: http://pubs.acs.org/doi/abs/10.1021/ja00051a040.
[40] J. Feldt, R.A. Mata, J.M. Dieterich, J. Chem. Inf. Model. 52 (2012) 1072. http://pubs.acs.org/doi/pdf/10.1021/ci2004219, URL: http://pubs.acs.org/doi/abs/10.1021/ci2004219.
[41] F. Aquilante, L. DeVico, N. Ferré, G. Ghigo, P.-Å. Malmqvist, P. Neogrdy, T.B. Pedersen, M. Pitok, M. Reiher, B.O. Roos, et al., J. Comput. Chem. (ISSN: 1096-987X) 31 (2010) 224. URL: http://dx.doi.org/10.1002/jcc.21318.
[42] M.-K.K. Wong, T.-C.C. Chan, W.-Y.Y. Chan, W.-K.K. Chan, L.L. Vrijmoed, D.K. O'Toole, C.-M.M. Che, Environ. Sci. Technol. (ISSN: 0013-936X) 40 (2006) 625. URL: http://view.ncbi.nlm.nih.gov/pubmed/16468412.

[43] R. Curci, A. Dinoi, M.F. Rubino, Pure Appl. Chem. (ISSN: 1365-3075) 67 (1995) URL: http://dx.doi.org/10.1351/pac199567050811.
[44] J.K. Crandall, R. Curci, L. D'Accolti, C. Fusco, Methyl(trifluoro-methyl)dioxirane, John Wiley & Sons, Ltd., Chichester, UK, ISBN: 0471936235, 2001, URL: http://dx.doi.org/10.1002/047084289x.rm267.pub2.
[45] L. Zou, R.S. Paton, A. Eschenmoser, T.R. Newhouse, P.S. Baran, K.N. Houk, J. Org. Chem. 78 (2013) 4037. URL: http://dx.doi.org/10.1021/jo400350v.
[46] K. Anderson, Dioxirane compounds and uses thereof (2013), wO Patent App. PCT/US2012/025,211,
URL: http://www.google.com/patents/WO2013122582A1?cl=en.
[47] R.D. Bach, J.L. Andres, A.L. Owensby, H.B. Schlegel, J.J.W. McDouall, J. Am. Chem. Soc. 114 (1992) 7207.
URL: http://chem.wayne.edu/schlegel/Pub_folder/143.pdf.
[48] A.D. Becke, J. Chem. Phys. 98 (1993) 5648.
URL: http://link.aip.org/link/JCPSA6/v98/i7/p5648/s1&Agg=doi.
[49] P.J. Stephens, F.J. Devlin, C.F. Chabalowski, M.J. Frisch, J. Phys. Chem. 98 (1994) 11623. URL: http://dx.doi.org/10.1021/j100096a001.
[50] R.H. Hertwig, W. Koch, Chem. Phys. Lett. (ISSN: 00092614) 268 (1997) 345. URL: http://dx.doi.org/10.1016/s0009-2614(97)00207-8.
[51] R. Ditchfield, W.J. Hehre, J.A. Pople, J. Chem. Phys. (ISSN: 0021-9606) 54 (2003) 724. URL: http://dx.doi.org/10.1063/1.1674902.
[52] W.J. Hehre, R. Ditchfield, J.A. Pople, J. Chem. Phys. (ISSN: 0021-9606) 56 (2003) 2257. URL: http://dx.doi.org/10.1063/1.1677527.
[53] M.W. Schmidt, K.K. Baldridge, J.A. Boatz, S.T. Elbert, M.S. Gordon, J.H. Jensen, S. Koseki, N. Matsunaga, K.A. Nguyen, S. Su, et al., J. Comput. Chem. (ISSN: 0192-8651) 14 (1993) 1347. URL: http://dx.doi.org/10.1002/jcc.540141112.
[54] M.S. Gordon, M.W. Schmidt, Advances in Electronic Structure Theory: GAMESS a Decade Later, Elsevier, Amsterdam, 2005, pp. 1167–1189.
[55] T.H. Dunning, J. Chem. Phys. (ISSN: 0021-9606) 90 (1989) 1007. URL: http://dx.doi.org/10.1063/1.456153.
[56] NIST computational chemistry comparison and benchmark database, NIST Standard Reference Database Number 101, Release 16a (2013), [online] http://cccbdb.nist.gov/, URL: http://cccbdb.nist.gov/.
[57] E.F.V. Scriven (Ed.), Azides and Nitrenes: Reactivity and Utility, Academic, New York, 1984.
[58] B. Iddon, O. Meth-Cohn, E.F.V. Scriven, H. Suschitzky, P.T. Gallagher, Angew. Chem., Int. Ed. Engl. 18 (1979) 900.
URL: http://dx.doi.org/10.1002/anie.197909001.
[59] F. Kotzyba-Hibert, I. Kapfer, M. Goeldner, Angew. Chem., Int. Ed. Engl. (ISSN: 0570-0833) 34 (1995) 1296.
URL: http://dx.doi.org/10.1002/anie.199512961.
[60] S.M. Mandel, J.A. KrauseBauer, A.D. Gudmundsdottir, Org. Lett. 3 (2001) 523. URL: http://dx.doi.org/10.1021/ol0068750.
[61] W. Carl Lineberger, W. Thatcher Borden, Phys. Chem. Chem. Phys. (ISSN: 1463-9076) 13 (2011) 11792. URL: http://dx.doi.org/10.1039/c0cp02786c.
[62] A. DeFusco, J. Ivanic, M.W. Schmidt, M.S. Gordon, J. Phys. Chem. A 115 (2011) 4574. URL: http://dx.doi.org/10.1021/jp112230f.
[63] P.N. Day, J.H. Jensen, M.S. Gordon, S.P. Webb, W.J. Stevens, M. Krauss, D. Garmer, H. Basch, D. Cohen, J. Chem. Phys. (ISSN: 0021-9606) 105 (1996) 1968. URL: http://dx.doi.org/10.1063/1.472045.
[64] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids, Oxford Science Publications (Oxford University Press), USA, ISBN: 0198556454, 1989, URL: http://www.worldcat.org/isbn/0198556454.
[65] M.J. Travers, D.C. Cowles, E.P. Clifford, G.B. Ellison, P.C. Engelking, J. Chem. Phys. (ISSN: 0021-9606) 111 (1999) 5349. URL: http://dx.doi.org/10.1063/1.479795.
[66] Jmol: an open-source Java viewer for chemical structures in 3D, [online] http://www.jmol.org/.