Performance analysis of the NWChem TCE for different communication patterns

Priyanka Ghosh
Dept. Of Computer Science
University of Houston
Houston, Texas 77004
pghosh06@cs.uh.edu

Jeff R.Hammond
Leadership Computing Facility
Argonne National Laboratory
Lemont, Illinois, 60439
jhammond@anl.gov

Sayan Ghosh
Dept. Of Computer Science
University of Houston
Houston, Texas 77004
sgo@cs.uh.edu

Barbara Chapman
Dept. Of Computer Science
University of Houston
Houston, Texas 77004
chapman@cs.uh.edu

***Abstract*—One-sided communication is a model that separates communication from synchronization, and has been in practice for over two decades in libraries such as SHMEM and Global Arrays (GA). GA is used in a number of application codes, especially NWChem, and provides a superset of SHMEM functionality that includes remote *accumulate*, among other features. Remote *accumulate* is an active-message operation that applies $y+ = a*x$ at the target rather than just $y = x$ (as in Put) which gives the programmer additional choices with respect to algorithm design. In this paper, we discuss and evaluate communication scenarios for dense block-tensor contractions, one of the mainstays of NWChem computation chemistry package. We show that apart from the classical approach involving dynamic scheduling of data-blocks for load-balancing, reordering one-sided Get and Accumulate calls affects the performance of tensor contractions on leadership class machines substantially. In order to understand why this reordering affects the performance, we develop a proxy application for the NWChem TCE (Tensor Contraction Engine) module. We utilize this proxy application to compare different implementations with a focus on communication.**

***Keywords*—NWChem, One-sided communication, Global Arrays, MPI-3, Tensor contractions**

I. Introduction

NWChem [14], [3] is one of the best-known applications that relies heavily upon one-sided communication, having employed the Global Arrays since the outset and never explicitly adopted the ubiquitous MPI model[1]. Global Arrays [9] is a PGAS (Partitioned Global Address Space) programming model in library form that includes a much richer set of functionality than SHMEM, which has existed for approximately the same amount of time. SHMEM provides a set of remote memory access (RMA) primitives that are implementable in hardware, thereby enabling efficient implementations that can – at least in theory – consume no processing resources except when processing elements (PEs) make SHMEM calls. On the other hand, GA's design exposes features that the programmer needs to implement mathematical algorithms found, for example, in quantum chemistry, which frequently include multidimensional array accesses and floating-point accumulate. Generalized noncontiguous access (i.e. beyond simple strided access) and remote floating-point accumulate are rarely implemented in network hardware (except when the "network" is shared-memory) and thus GA consumes processing resources in the background - frequently in the form of a communication helper thread but occasionally as remote interrupts - that can have a nontrivial impact on performance in some cases [5]. However, the overall application performance benefits despite this overhead because of greater algorithmic flexibility in the application design, since, for example, eliminating remote accumulate in favor of remote writes reduces parallelism because, unlike the former, the latter cannot occur concurrently on the same remote buffer within a synchronization epoch.

Parallel performance optimization of NWChem is much like any other code in most respects; one attempts to reduce load-imbalance [13], [1], [10], use caching to avoid communication if not a memory bottleneck and reduce the cost of remote data lookup [7], etc. Just as in MPI programs, one can trade a set of communication operations for another equivalent set, for example, in MPI one pad arrays in FFT to replace `MPI_Alltoallv` with `MPI_Alltoall`, which is more efficient if the padding overhead is low; on the other hand, trading `MPI_Alltoallv` for many send-recv calls will be more efficient in the sparse case (many processes send no data). Transforming communication patterns are not always about reducing data movement, of course; reducing synchronization overhead is essential for scalable algorithms.

In GA programs, there are two types of synchronization overheads, explicit and implicit. Obviously, explicit synchronization overheads are the well-known ones, e.g. global synchronization via a barrier (as in `ga_sync`). Implicit synchronization is required when issuing a Get, for example, as the request for remote data must be delivered to the remote node[2], acted upon and the data delivered back to the initiator. On the other hand, Put and remote accumulate (Acc) may complete locally without any remote activity. Additional latency for Get with respect to Put or Acc can arise from contention on the network, waiting in the remote active-message queue, and the cost of the active-message itself, which might entail a strided memory copy that crosses multiple levels of the memory hierarchy. Of course, Put and Acc see all of these same effects but the cost is mostly hidden from the application, at least w.r.t. direct measurement of their costs, because the initiator proceeds without waiting on their remote completion. In aggregate, however, the sum of the remote processing costs

[1] The exception to this statement is not relevant to this paper.

[2] Different node resources may be involved in processing these requests depending on the implementation strategy and the nature of the request itself; contiguous requests may be processed in hardware while noncontiguous requests often require processor intervention to pack messages to be large enough to saturate the network bandwidth.

are visible unless the associated resources would otherwise be idle. For this reason, Acc is more expensive than Put, since it requires floating-point computation. What is not clear is whether - in terms of aggregate parallel performance - one should favor Get or Acc. One of the goals of this paper is to determine the answer to this question experimentally.

One of the more popular methods in NWChem is the quantum many-body method known as coupled cluster theory (CC), which is used heavily for solving chemical problems requiring quantitative accuracy for chemical bond-breaking, e.g. combustion. Given its steep computational cost – $O(N^7)$ in the case of the most commonly used CC variant, CCSD(T) – simulations running on thousands of cores for many hours are rather common, hence even modest performance improvements in the scalability and efficiency of these methods pay off significantly. Given that much of the floating-point computation in CC simulations is performed using the BLAS3 subroutine `DGEMM`, which usually runs at more than 80% of peak for the dimensions used in NWChem, the most obvious place for optimization in NWChem CC codes is in the communication, synchronization and load-balancing.

Past approaches to improve performance involve efficient strategies to reduce the amount of load imbalance in the CCSD module [10]. In this paper, our focus is the design space of the Tensor Contraction Engine (TCE) module to evaluate the performance behavior observed for CCSD block tensor contractions when restructuring Get and Acc calls. We consider two types of optimizations: (1) reordering of the loops over tiles to trade Get calls for Acc calls and (2) reduction in Get calls by fetching multiple blocks at a time into a cache. Because of the complexity of individual tensor contractions in NWChem, which support a range of dimensions (up to 8-dimensional arrays) with both permutation symmetry and block sparsity (which arises from point-group symmetry - see [10] for details) and contain a number of optimizations that are opaque to non-chemists, we implemented a stripped down proxy application that exposes only the essential features of an individual tensor contraction implemented using the TCE template. The proxy considers only the cases of 2-dimensional arrays[3] and uses regular tiling, which reduces some of the issues associated with load-balancing. Thus, the proxy is most useful for understanding the trade-offs associated with restructuring the communication operations, which is exactly the purposes for which it was designed. We did not implement a proxy version of optimization (2) since it was not necessary to understand the results for that case.

The primary contributions of this paper are: (1) design and implementation of two new variants of the NWChem TCE tensor contraction template that are demonstrated empirically to be significantly faster than the original for real-world chemistry problems on the primary hardware platform for NWChem (x86/InfiniBand); (2) development of a proxy application to model the different communication pattern found in one of these variants; and (3) implementation of the proxy application in the original GA programming model and MPI (3.0 standard [8]) to evaluate any potential differences between the communication infrastructure therein. The differences in the behavior of the full NWChem application and the proxy demonstrate the shortcomings of the proxy and reveal the downsides of eliminating domain-specific application complexity.

[3] Obviously, a tensor contraction involving 2-dimensional arrays is just a matrix-matrix multiplication and optimal algorithms for these are known, but we are trying to model arbitrary dimensionality and the benefit to the proxy of supporting this is not justified given the additional complexity required to implement it explicitly.

II. Implementation & Design

The Tensor Contraction Engine (TCE) [6], [2] is a project to automate the derivation and parallelization of quantum many-body methods such as CC. A large number of procedures are necessary because of many different types of contractions possible between tensors of various rank. All tensors are decomposed using tiling, where each tile falls into a single symmetry class (see [6] for details) and these tiles define the granularity of all tensor contractions.

A. Algorithms implemented in NWChem

Algorithm 1 provides an overview of the default implementation of a distributed tensor contraction in TCE. *NXTVAL* represents the centralized dynamic load balancer inherited from TCGMSG, a pre-MPI communication library which queries a global counter and runs over all possible tasks in a given contraction. The *Symmetry* function is a condensation of a number of logical tests in the code that determine whether a particular tile will be nonzero. In Algorithms 1, 2 and 3, the indices given for the local buffer contraction are the tile indices where each tile index represents a set of contiguous indices and where each tile is grouped in such a way such that the symmetry properties of all its constitutive elements are identical.

Algorithm 1 illustrates the *Original* (default) version of the tensor contraction template, wherein all the *Get*'s are performed in the innermost loop repeatedly for fetching every tile into local buffers. However this approach entails redundant calls to *TCE_Get* since the same tile is fetched multiple times during subsequent iterations. The calls to *GA_Accumulate* are however sparse due to the aggregation of dense inner loop computations satisfying a set of symmetry conditions into a single task. *TCE_Get* is shorthand for a small number (between 1 and 8) of *GA_Get* operations combined with the accumulation of the resulting buffers from these onto a single buffer, as described in [7].

Algorithm 2 illustrates the *Inverted* version of the TCE-CCSD method where the number of calls to *Get*'s are drastically reduced by transferring the *get* call to the outer loop. Therfore a tile once fetched into the local buffer *b*, is reused in all computations of the dense inner loop satisfying the set of symmetry conditions. The penalty on the hand mandates the need for excessive number of calls to *accumulate*, after it is moved inside the innermost loop in order to maintain the symmetry criterion.

In quantum chemistry applications such as NWChem a usually large molecular system lacking any spatial symmetry produces set of tiles which align well to the number of processors and can be used to obtain high scalability. In such a scenario, we encounter lesser load imbalance among tasks owing to the proper distribution of the dense computations and thus stand to gain performance benefit if we were to reduce

ALGORITHM 1: Pseudocode for *Original* version of TCE-CCSD implementation

```
Tiled Global Arrays: A, B, C
Local buffers: a, b, c;
forall the h1, h2, p3, p4 ∈O,V tiles do
    if NXTVAL(my_pe)=True then
        if Symmetry(h1,h2,p3,p4)=True then
            Allocate c for C(h1,h2,p3,p4) tiles
            for p5, p6 ∈V tiles do
                if Symmetry(h1,h2,p5,p6)=True then
                    if Symmetry(p5,p6,p3,p4)=True then
                        TCE_Get A(h1,h2,p5,p6) into a
                        TCE_Get B(p5,p6,p3,p4) into b
                        c(h1,h2,p3,p4) += a(h1,h2,p5,p6) *
                        b(p5,p6,p3,p4)
                    end
                end
            end
            GA_Acc c into C(h1,h2,p3,p4)
        end
    end
end
```

ALGORITHM 2: Pseudocode for *Inverted* version of TCE implementation

```
Tiled Global Arrays: A, B, C
Local buffers: a, b, c;
forall the p3, p4, p5, p6 ∈V,V tiles do
    if Symmetry(p3,p4,p5,p6)=True then
        if NXTVAL(my_pe)=True then
            TCE_Get B(p5,p6,p3,p4) into b
            for h1, h2 ∈O tiles do
                if Symmetry(h1,h2,p3,p4)=True then
                    if Symmetry(h1,h2,p5,p6)=True then
                        Allocate c for C(h1,h2,p3,p4) tiles
                        TCE_Get A(h1,h2,p5,p6) into a
                        c(h1,h2,p3,p4) += a(h1,h2,p5,p6) *
                        b(p5,p6,p3,p4)
                        GA_Acc c into C(h1,h2,p3,p4)
                    end
                end
            end
        end
    end
end
```

ALGORITHM 3: Pseudocode for *Cache* version of TCE implementation

```
Tiled Global Arrays: A, B, C
Local buffers: a, b, c;
Local Hash Table: Htable;
forall the h1, h2, p3, p4, p5, p6 ∈O,V,V tiles, unroll U do
    if NXTVAL(my_pe)=True then
        forall the h1i, h2i, p5i, p6i ∈U,U tiles do
            if Symmetry(h1i,h2i,p5i,p6i)=True then
                TCE_Get A(h1i,h2i,p5i,p6i) and save into Htable
            end
        end
        forall the p3i, p4i, p5i, p6i ∈U,U tiles do
            if Symmetry(p3i,p4i,p5i,p6i)=True then
                TCE_Get B(p5i,p6i,p3i,p4i) and save into Htable
            end
        end
        forall the h1i, h2i, p3i, p4i ∈U,U tiles do
            Allocate c for C(h1i,h2i,p3i,p4i) tiles
            for p5i, p6i ∈U tiles do
                if Symmetry(h1i,h2i,p3i,p4i)=True then
                    if Symmetry(h1i,h2i,p5i,p6i)=True then
                        Fetch block(h1i,h2i,p5i,p6i) from Htable
                        into a
                        Fetch block(p5i,p6i,p3i,p4i) from Htable
                        into b
                        c(h1i,h2i,p3i,p4i) += a(h1i,h2i,p5i,p6i) *
                        b(p5i,p6i,p3i,p4i)
                    end
                end
            end
            GA_Acc c into C(h1i,h2i,p3i,p4i)
        end
    end
end
```

the cost of communication, by minimizing the number of calls to *get*'s and *accumulate*'s. Algorithm 3 illustrates such an approach wherein we strip the outer loop and save the blocks obtained from a *get* call for multiple reuse later. This approach (*Cache* version) guarantees fewer calls to *get* compared to both the *Original* and *Inverted* versions as well as fewer calls to *accumulate* in comparison to the *Inverted* version.

B. TCE Proxy applications

We designed a proxy application replicating the TCE module functionality for both the *Original* and *Inverted* versions. We implemented this using two programming models namely Global Arrays and MPI. Apart from revealing software overheads between MPI and Global Arrays versions of the block tensor contraction kernel, the RMA implementation of MPI is fundamentally different to that of ARMCI and by extension, Global Arrays. As the RMA implementation of MPI has substantially improved, we expected better results for the "Original" version (lesser "accumulates" with local completion semantics [passive target synchronization]) for most of the test data sizes, than the "inverted" version, which requires a request handle for waiting (local completion) on outstanding "accumulates".

The "inverted" version incorporates a double-buffering scheme, wherein 2 local buffers for the same array are maintained, one for communication and the other for computation. Data is transferred using the communication buffer, and local computations proceed with the compute buffer (a wait is issued prior to using the compute buffer, to ascertain completion of previous task). The buffers are swapped at the end of an iteration. MPI request based accumulate (`MPI_Raccumulate` with `MPI_Wait`) and ARMCI non-blocking operation (`ARMCI_NbAccS` with `ARMCI_Wait`) are used for taking advantage of this overlap in computation and communication.

III. Experimental Results

We have performed all our experiments on an InfiniBand cluster Tukey, a 96-node 16-core 2-way SMP AMD processor with 64 GB memory per node at Argonne National Laboratory. The system is running Linux kernel 2.6.32 (x86_64). NWChem was compiled with GCC 4.4.6, because of the heavy reliance on BLAS for floating-point-intensive kernels, for which we employ GotoBLAS 2 1.13[4] library. For the proxy applications (GA/MPI), BLAS from ATLAS library[15] is used. The high-performance interconnect is InfiniBand QDR with a theoretical throughput of 4 GB/s per link and $\sim$ 2 μs latency. The communication libraries used were ARMCI from Global Arrays 5.1, which is heavily optimized for InfiniBand,

with MVAPICH2 1.8 (for NWChem and GA proxy) and MVAPICH2-X 2.a (for MPI proxy). We have launched all our NWChem experiments with 8 MPI processes per node, whereas for the proxy applications, we have 4 MPI processes per node.

A. *Full application tests*

Figures 1 and 2 represent the comparison in terms of performance obtained for a 8-H_2O and 9-H_2O CCSD simulation respectively, with the aug-cc-pvdz basis running on the Tukey cluster. CC simulations can be categorized as symmetrically sparse or asymmetrically dense. We find the problems falling in the latter category, (characteristic to the water molecule), a more suitable candidate to perform the cache optimization. In both figures we notice the strong scalability of the *Inverted* version in comparison to the *Original* version. This is attributed to fewer *Get* operations performed in the *Inverted* version. Since *Get*'s in NWChem have to satisfy several symmetry constraints, it is a far costlier operation compared to *accumulate*. The *Cache* version accounts for a 27% reduction in the total number of *Get*'s compared to *Inverted* and a 72% reduction compared to *Original* (as seen in Table I). This contributes towards an average performance improvement of 20% and 10% with respect to *Original* and *Inverted* respectively due to the reduction in the overall network communication.

The percentage of improvement is subject to change based on the size and symmetry of the molecule being used in the simulation. With smaller molecules, which lead to fewer tasks, the dynamic load balancer (NXTVAL) will add some overhead for the *Cache* version and in case of molecules with high symmetry, we may not encounter a reduction in the total number of *Get*'s.

Input Data Molecule		**Total Number**	
8-H_2O		Get	Accumulate
	Original	84700	682
	Inverted	32065	71632
	Cache	23353	21582
9-H_2O		Get	Accumulate
	Original	119448	810
	Inverted	45072	101196
	Cache	32904	25650

TABLE I: Results comparing the number of *gets* and *accumulates* encountered for the three TCE-CCSD implementations

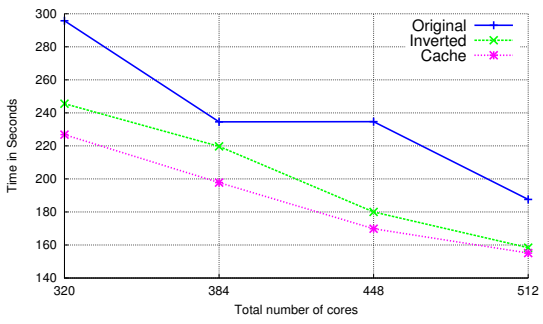

Fig. 1: Results comparison for water w8 aug-cc-pvdz

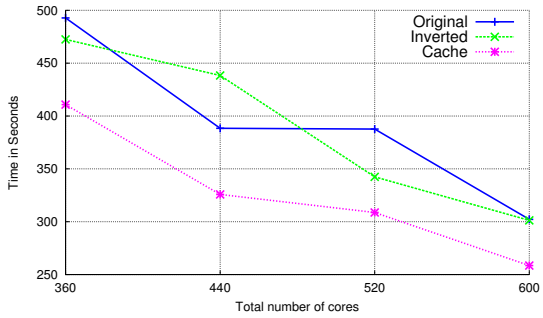

Fig. 2: Results comparison for water w9 aug-cc-pvdz

Table I draws a comparison based on the number of *gets* and *accumulates* recorded for the various implementations of TCE-CCSD module in NWChem. These results have been acquired using a TCE profiling interface (TPI) constructed within the TCE framework.

B. *Proxy applications using Global Arrays and MPI*

1) Global Arrays: The results obtained for the proxy application using Global Array depicted in Figures 3 and 4 representing square matrix sizes of 24K and 32K respectively, clearly indicate that the *Original* version surpasses the *Inverted* version in terms of performance. In contrast to NWChem this behavior is expected, since the *Get* operations in the proxy application are computationally less expensive in comparison to NWChem resulting in *Accumulate*'s becoming more costly. Since the *Original* version performs the least number of *accumulates* it clearly wins in terms of performance. However the *Original* version demonstrates weaker scalability compared to *Inverted* especially for cases with larger block sizes as seen in Figure 4 with block size of 1200. We suspect this is owing to the creation of fewer number of tasks in case of *Original*, where the lack of core count saturation results in higher load imbalance.

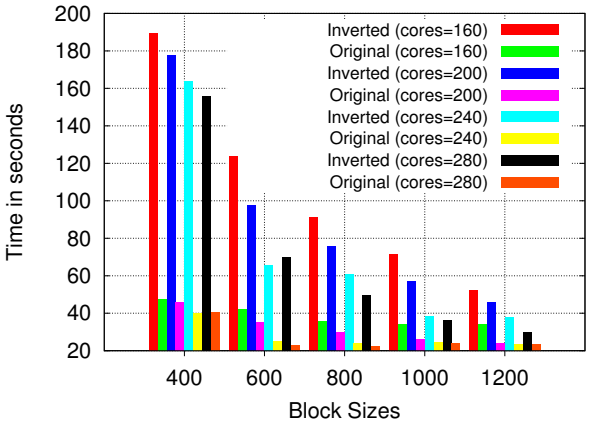

Fig. 3: 24K square matrix with ppn=4 and nodes=40,50,60 and 70 using GA

2) MPI: Since the domain decomposition technique for MPI proxy-app versions rely on dimensions being perfectly divisible by the total number of processes, so we have chosen only the cases which satisfies this criteria. The performance

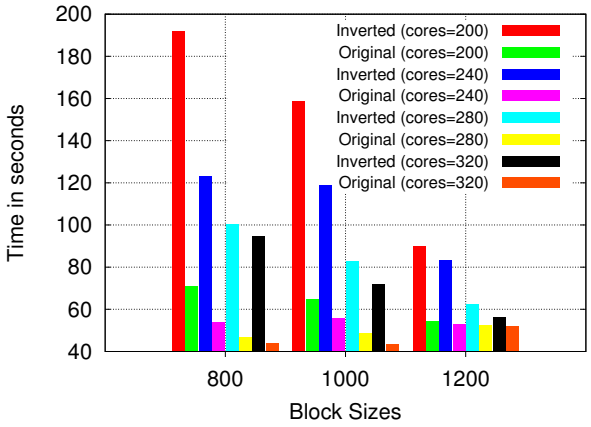

Fig. 4: 32K square matrix with ppn=4 and nodes=50,60,70 and 80 using GA

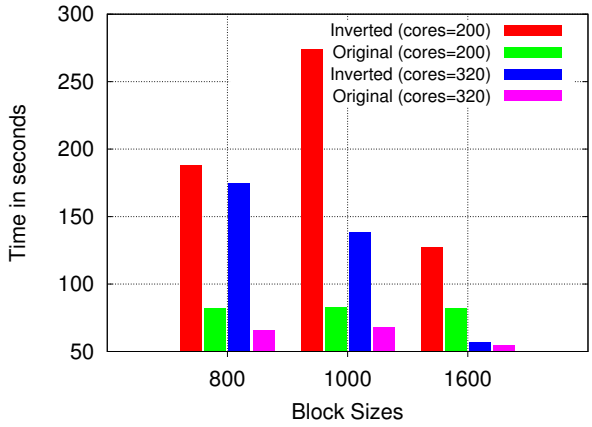

Fig. 6: 32K square matrix with ppn=4 and nodes=50 and 80 using MPI

of *Inverted* versions are sub-par compared to that of *Original*, which could be attributed to a greater number of *accumulates*. However, the *Inverted* versions for larger processes were observed to be better in terms of scalability. This pattern is observed from execution on 240 processes for 24K matrices as shown in Figure 5 and 32K matrices using 320 processes in Figure 6. This indicates that for larger data sizes (with coarser block sizes) on proportionally large number of processes benefits from the overlap of computation and communication. However, for the *Original* case, we observe from Figures 5 and 6 that execution times remain fairly consistent with different block sizes, and transitions between small block sizes (like 600 to 1000 in 24K matrices shown in Figure 5) is almost negligible. In this case too, large block sizes (and a larger number of processes) have a positive impact on performance, as we could observe from Figure 6 (12 secs difference between block sizes 1600 and 800 on 320 processes for 32K matrices), as opposed to a trivial change in performance from 800-1600 block sizes on 200 processes for the same case.

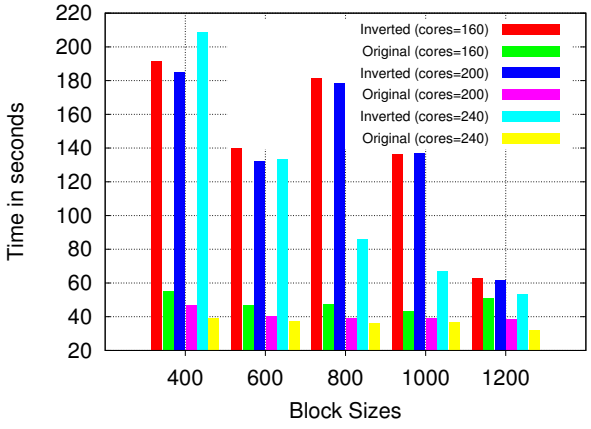

Fig. 5: 24K square matrix with ppn=4 and nodes=40,50 and 60 using MPI

IV. Related Work

Fundamentally different algorithms for distributed-memory tensor contractions have been explored in the Cyclops Tensor Framework (CTF) [12]. The relative merits of the CTF approach and the TCE approach depend upon the details of the architecture and the degree of irregularity (e.g. block-sparsity) in the computation. Sadayappan and coworkers have attempted to bring together the best of CTF and TCE in the CAST project [11], but the lack of integration into NWChem means we cannot compare directly with our full application results.

V. Conclusions and Future Work

We identified two new implementations of the TCE template that improve the performance of NWChem by more than 10%. One of these is based upon inverting communication to reduce Get in favor of Acc while the other reduces Get using caching. To understand how the different communication patterns affect performance, we developed a simple proxy application and tested it using both GA and MPI. However, the simple proxy was unable to reproduce the quantitative results of the full application, although some of the basic trends were the same. We are developing more complex proxies with the intent of reconciling their behavior to the full application. Once more accurate proxies are developed, we can employ them to understand a variety of one-sided programming models, including GA, MPI, OpenSHMEM and PGAS languages such as UPC and CAF. We will also expand our experimental studies to other architectures and to many thousands of cores.

VI. Acknowledgement

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] Edoardo Aprà, Alistair P. Rendell, Robert J. Harrison, Vinod Tipparaju, Wibe A. de Jong, and Sotiris S. Xantheas. Liquid water: obtaining the right answer for the right reasons. In *Proceedings of the ACM/IEEE SC Conference on High Performance Networking and Computing*, pages 1–7, New York, NY, USA, 2009. ACM.

[2] Alexander A. Auer, Gerald Baumgartner, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[3] E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma, M. Valiev, H. J. J. van Dam, D. Wang, E. Aprà, T. L. Windus, J. Hammond, J. Autschbach, P. Nichols, S. Hirata, M. T. Hackler, Y. Zhao, P-D Fan, R. J. Harrison, M. Dupuis, D. M. A. Smith, J. Nieplocha, V. Tipparaju, M. Krishnan, A. Vazquez-Mayagoitia, Q. Wu, T. Van Voorhis, A. A. Auer, M. Nooijen, L. D. Crosby, E. Brown, G. Cisneros, G. I. Fann, H. Früchtl, J. Garza, K. Hirao, R. Kendall, J. A. Nichols, K. Tsemekhman, K. Wolinski, J. Anchell, D. Bernholdt, P. Borowski, T. Clark, D. Clerc, H. Dachsel, M. Deegan, K. Dyall, D. Elwood, E. Glendening, M .Gutowski, A Hess, J. Jaffe, B. Johnson, J. Ju, R. Kobayashi, R. Kutteh, Z. Lin, R. Littlefield, X Long, B. Meng, T. Nakajima, S. Niu, L. Pollack, M Rosing, G Sandrone, M. Stave, H. Taylor, G. Thomas, J. van Lenthe, A. Wong, and Z. Zhang. NWChem, a computational chemistry package for parallel computers, version 6.0, 2010.

[4] Kazushige Goto. Gotoblas. *Texas Advanced Computing Center, University of Texas at Austin, USA. URL¡ http://www. otc. utexas. edu/ATdisplay. jsp*, 2007.

[5] Jeff R. Hammond, Sriram Krishnamoorthy, Sameer Shende, Nichols A. Romero, and Allen D. Malony. Performance characterization of global address space applications: a case study with NWChem. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2011.

[6] So Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[7] Karol Kowalski, Jeff R. Hammond, Wibe A. de Jong, Peng-Dong Fan, Marat Valiev, Dunyou Wang, and Niranjan Govind. Coupled cluster calculations for large molecular and extended systems. In Jeffrey R. Reimers, editor, *Computational Methods for Large Systems: Electronic Structure Approaches for Biotechnology and Nanotechnology*. Wiley, 2011.

[8] MPI Forum. MPI: A message-passing interface standard. Version 3.0., November 2012.

[9] Jarek Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A portable "shared-memory" programming model for distributed memory computers. In *Supercomputing (SC)*, 1994.

[10] David Ozog, Jeff R. Hammond, James Dinan, Pavan Balaji, Sameer Shende, and Allen Malony. Inspector-executor load balancing algorithms for block-sparse tensor contractions. In *International Conference on Parallel Processing (ICPP)*, October 2013.

[11] Samyam Rajbhandari, Akshay Nikam, Pai-Wei Lai, Kevin Stock, Sriram Krishnamoorthy, and P. Sadayappan. Framework for distributed contractions of tensors with symmetry. Preprint, Ohio State University, 2013.

[12] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. Cyclops tensor framework: reducing communication and eliminating load imbalance in massively parallel contractions. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[13] Tjerk P. Straatsma and James Andrew McCammon. Load balancing of molecular dynamics simulation with NWChem. *IBM Systems Journal*, 40(2):328–341, 2001.

[14] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[15] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27. IEEE Computer Society, 1998.