Scaling NWChem with Efficient and Portable Asynchronous Communication in MPI RMA

Min Si,[*] Antonio J. Peña,[†] Jeff Hammond,[‡] Pavan Balaji,[†] Yutaka Ishikawa[§]

[*]University of Tokyo, Japan *msi@il.is.s.u-tokyo.ac.jp*
[†]Argonne National Laboratory, USA {*apenya, balaji*}*@mcs.anl.gov*
[‡]Intel Labs, USA *jeff_hammond@acm.org*
[§]RIKEN AICS, Japan *yutaka.ishikawa@riken.jp*

***Abstract*—NWChem is one of the most widely used computational chemistry application suites for chemical and biological systems. Despite its vast success, the computational efficiency of NWChem is still low. This is especially true in higher accuracy methods such as the CCSD(T) coupled cluster method, where it currently achieves a mere 50% computational efficiency when run at large scales. In this paper, we demonstrate the most computationally efficient scaling of NWChem CCSD(T) to date, and use it to solve large water clusters. We use our recently proposed process-based asynchronous progress framework for MPI RMA, called Casper, to scale the computation on water clusters at near-100% computational efficiency on up to 12288 cores.**

***Keywords*-NWChem; Global Arrays; CCSD(T); MPI RMA; one-sided; asynchronous progress**

I. Introduction

NWChem [1] quantum chemistry application is one of the most widely used computational chemistry application suites and offers extensive capabilities for large-scale simulations of chemical and biological systems. However, such chemistry applications are often operate on large data sets that easily exceed the capacity of a single node, making resource sharing across nodes necessary. The Global Arrays (GA) programming model [2] was developed to address the large memory requirements in the NWChem and also has been widely used in other scientific domains in recent years [3] [4]. GA provides global-view access to multidimensional arrays distributed across the memories of multiple nodes through asynchronous one-sided operations such as *put* and *get*.

Application developers are increasingly looking at such programming models for their large-scale applications because of the more natural mapping to certain classes of irregular algorithms as well as one-sided networks (e.g., RDMA). In quantum chemical many-body methods, the dominant cost is often block-sparse tensor contractions, which in NWChem are implemented in terms of dense matrix operations (i.e., BLAS) performed driven by the *get–compute–update* pattern common to many GA applications. Such application motifs critically depend on the asynchronous completion of one-sided operations in order to hide the overhead of frequent data movements within intensive computations.

Dinan et al. [5] introduced a *portable* GA implementation by interacting with the MPI runtime system, the *de facto* standard communication interface for distributed-memory systems. This GA implementation is built on top of the MPI one-sided communication primitives (also known as remote memory access or RMA), whose one-sided semantics are similar to those of the high-level GA programming model. The RMA semantics allow a process to access memory regions of other processes without the target process explicitly needing to receive or process a message. Such a mode, however, is not guaranteed to be completely *asynchronous* in the sense that some MPI implementations still require the *remote target* to make MPI calls in order to make progress on those operations that cannot be offloaded to the underlying hardware. The lack of asynchronous progress extremely limits the performance and *scalability* of GA-based applications, often dominated by intensive computation, and results in significant RMA communication delays.

In this paper we analyze the impact of software-handled MPI RMA communication on NWChem. The parallel efficiency can be artificially high if it is calculated by using an inefficient base execution (i.e., performs inefficient communication). Thus, instead of the parallel efficiency, we use Equation 1 to evaluate the computational efficiency $E(N)$ of an execution:

$$E(N) = \frac{T_{comp}/T_N}{N} = \frac{T_{\alpha_{comp}} \times \alpha}{T_N \times N} \tag{1}$$

in which T_{comp} is the computation time on single cores and T_N is the total execution time including both communication and computation on N cores. In practice, however, most NWChem problems do not fit on a single core because of large memory requirements, thus we determine the T_{comp} by using $T_{\alpha_{comp}} \times \alpha$ where α is the minimum number cores measured for a given problem size and $T_{\alpha_{comp}}$ is the time spent in computation on α cores. A high computational efficiency means the overhead of communication is relatively low, thus the high parallel efficiency is meaningful.

Although NWChem provides highly efficient parallel algorithms for a variety of chemistry simulation problems especially for some computation-intensive methods such as CCSD(T), significant performance scaling bottlenecks have been observed in the water molecule problems on the time-consuming (T) portion, achieved as low as 50% computational efficiency, because of the lack of asynchronous progress in

RMA communication.

To address the scaling issue, we utilize "Casper," an efficient process-based asynchronous progress model for MPI RMA on multicore and many-core architectures [6]. Casper allows users to dedicate an arbitrary number of processor cores as background "ghost processes" in order to effectively perform RMA communication progress asynchronously on behalf of the computing processes, thus providing an efficient solution to help with asynchronous communication in GA-based applications without the overuse of computational resources.

Using the Casper framework, we scale the computation on water molecule problems at close to 100% computational efficiency. We show that this performance is maintained up to $(H_2O)_{21}$ running on 12288 cores. Although we use NWChem as our test case, we believe Casper's benefits are generally applicable to most GA-based applications on many-core architectures.

II. Background

We begin this section with an overview of the communication model in Global Arrays. Next we present the NWChem application.

A. Global Arrays

The Global Arrays toolkit [2] is a partitioned global address space library that provides users with distributed dense arrays that can be accessed through one-sided operations. It is widely used in many computational chemistry applications to address their large memory requirements. In the GA model, large array structures are stored in memory across multiple nodes. A process can access remote subarrays through one-sided *get*, *put*, and *accumulate* operations.

Common programming approaches in GA-based applications follow *get–compute–update* patterns. For example, as shown in Algorithm 1, when multiple processes are involved in a large matrix-matrix multiplication (MMM) $C = A \times B + C$, each process first gets submatrix a and submatrix b from global arrays that are potentially located in the memory spaces of remote processes, next performs a local matrix multiplication $c = a \times b + c$, and then updates submatrix c back to the global memory, accumulating the values. In order to achieve better scalability on modern parallel systems, the large matrix is partitioned among hundreds or thousands of processes, resulting in numerous one-sided operations.

B. NWChem

NWChem [1] is one of the most popular computational chemistry application suites for chemical and biological systems. It is developed based on the GA model because of the large memory needs that require memory sharing across multiple nodes. The coupled cluster (CC) theory is one of the most widely used approaches in quantum chemistry for computing electron correlation in atoms and molecules, that is, for the solution of the electronic Schrödinger equation with arbitrary accuracy requirements. NWChem provides highly efficient parallel implementations for a variety of complicated CC methods through the Tensor Contraction Engine (TCE). The "gold standard" *coupled cluster with singles and doubles and perturbative triples* method, known as CCSD(T), is one of the most accurate CC methods applicable to large molecules to date. It is particularly useful for describing accurate noncovalent interaction energies.

Algorithm 1: $C_J^I = A_K^I \times B_J^K + C_J^I$ matrix–matrix multiplication in get–compute–update mode.

```
Global Arrays: A, B, C;
Local Buffers : a, b, c;
for $i \in I$ do
    for $j \in J$ do
        for $k \in K$ do
            GET $a(i,k)$ from $A$;
            GET $b(k,j)$ from $B$;
            Compute $c(i,j) = a(i,k) \times b(k,j) + c(i,j)$;
        UPDATE $c(i,j)$ to $C$;
```

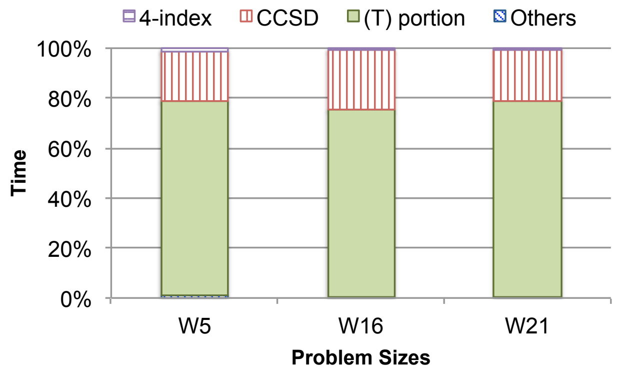


Fig. 1. Analysis of CCSD(T) internal steps in varying W_n with pVDZ.

The CCSD(T) method performs a complex set of multidimensional array computations organized in three internal steps: four-index transformation, CCSD iteration, and the noniterative (T) portion. To understand the performance characteristics of CCSD(T), we compared the time consumed by each step on our experimental platform (see Section VI) for three water molecule $(H_2O)_n$ problems ($n = 5, 16, 21$, denoted as W_n) with double-zeta basis sets (pVDZ). As shown in Figure 1, the (T) portion consistently dominates the entire cost of CCSD(T) by close to 80%, and the CCSD iteration takes the other 20%; the four-index transformation and other internal steps represent less than 3% of the execution time.

To improve the scalability properties of NWChem on top of MPI, we focus on optimizing the most time-consuming step: (T), a computation-intensive stage that follows the typical *get–compute–update* approach containing large MMM operations (*compute*) with numerous one-sided operations (*get*) and a number of reduce operations (*update*).

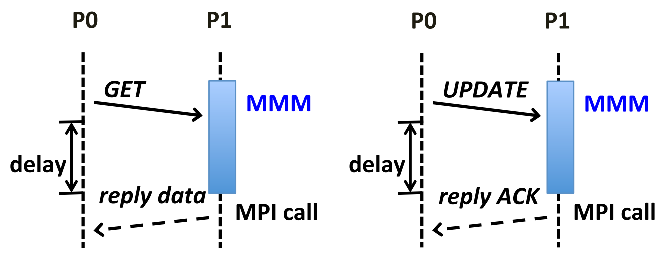


Fig. 2. Software-handled RMA in a GA environment.

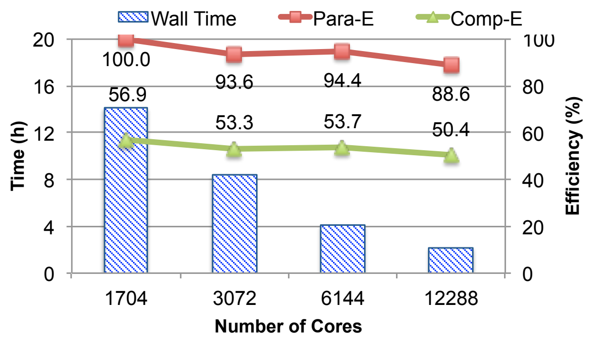


Fig. 3. Strong-scaling efficiency of (T) for W_{21} with pVDZ.

III. The Challenge

As noted in Section I, RMA operations are not always completely asynchronous. Many MPI implementations still require the target process to make MPI calls in order to process RMA operations issued on it as a target. For example, on RDMA-supported networks such as InfiniBand, contiguous put/get operations can be fully handled in hardware; however, complex noncontiguous accumulate operations, such as an accumulate operation on a 3D subarray, still have to be performed in software within the MPI implementation. Moreover, some MPI implementations, such as MPICH [7] or Cray MPI [8], still implement all RMA operations in software by default. Such software-handled RMA operations cannot complete at the target without explicit MPI calls. Consequently, arbitrarily long delays can occur if the target process is performing intensive computations such as MMM operations, as depicted in Figures 2(a) and 2(b).

The communication delay caused by software-handled RMA can be large in the computation-intensive (T) calculation in NWChem and can result in significant performance scaling bottlenecks, especially for large-scale problems. Figure 3 shows the strong-scaling performance with a comparison of the parallel efficiency (Para-E) and the computational efficiency (Comp-E) for W_{21} with double-zeta basis sets in NWChem on our experimental platform. As shown in the figure, although the parallel efficiency is maintained at near-100%, its computational efficiency is only about 50% for a range of core counts, resulting in up to 6 hours of additional run time for the 1,704-core case.

This low computational efficiency is caused by the lack of asynchronous progress for software-handled MPI RMA operations. While researchers have devised several approaches to ensure the asynchronous completion of operations, not all of these are efficient for GA-based applications. Indeed, performance may be degraded in various ways. In Section IV we discuss the impact of transitional asynchronous progress techniques. We present our solution in Section V and a complete evaluation in Section VI.

IV. Related Approaches

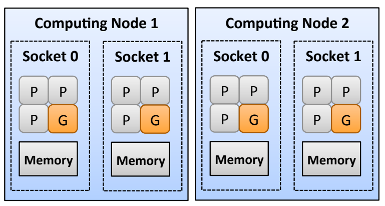
Traditional approaches to ensure asynchronous progress for MPI communication have relied on thread-based or interrupt-based models. These, however, feature performance limitations when deployed on modern architectures.

A. Thread-Based Approach

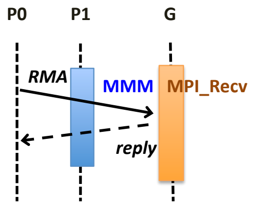
In the thread-based model, each MPI process utilizes a background thread in order to handle incoming messages from other processes. This model is used by many MPI implementations, including MPICH [7], MVAPICH [9], and Intel® MPI [10]. While being a generic approach for various MPI communication models, it raises performance concerns. A background thread can make progress for only the MPI process that spawned it. Thus, such a model requires deploying at least as many background threads as MPI processes. On current MPI implementations, where the progress engine polls repeatedly the network for incoming messages, this approach can waste half the computing resources. Oversubscribing the computing cores with the MPI processes along with the helper threads may result in further performance degradation. These issues especially impact the performance and scalability of applications with intensive computation (e.g., CCSD(T) in NWChem). Furthermore, this model forces MPI implementations to implement multithreaded safety, which may bring further bottlenecks because of thread synchronization requirements [11].

B. Interrupt-Based Approach

In the interrupt-based model, hardware interrupts are issued to awaken a thread in order to process the incoming RMA messages. This model is used by Cray MPI [8] when RMA uses the DMAPP conduit (not currently the default); in this case, the interrupt wakes up a kernel thread. MPI on Blue Gene/P [12, Chapter 7] and Blue Gene/Q [13] use special hardware to cause a context switch that cause a special thread to wake up and drive the network when a message arrives; in this case, the thread is a user thread, which allows for arbitrary code to run, unlike a kernel thread. While interrupt-driven asynchrony does not require dedicated resources the way polling threads do, handling interrupts on cores that are otherwise devoted to computation causes those cores to stop computing temporarily and leads to cache pollution. The overhead can be high in large-scale GA-based applications that rely on massive parallelism and frequent RMA operations [14].



(a) Topology-aware core deployment.



(b) Asynchronous progress.

Fig. 4. MPI RMA with Casper.

V. Solution

Our recent work introduced Casper, a process-based asynchronous progress model for MPI RMA communication on multicore and many-core architectures [6]. In this paper we use Casper to address the communication challenges existing in NWChem on top of a portable communications layer that effectively prevent its practical use in solving large problems.

Unlike traditional approaches, the rationale behind Casper is centered on the notion that since more and more cores are embedded into computing systems, some of these cores may not always be busy performing computation, and hence it may be more efficient to dedicate some of them to perform progress on asynchronous communication. Figure 4(a) shows an example of a topology-aware core deployment in Casper, including user processes (P) and "ghost processes" (G).

Casper is designed as an external library through the PMPI name-shifted profiling interface of MPI, which allows Casper to transparently link with any MPI implementation by overloading the necessary MPI functions. The central idea of Casper is to keep aside a small user-specified number of cores as "ghost processes" at the MPI initialization stage. When the user process (P) tries to allocate a remotely accessible memory window, Casper intercepts the call and maps that memory into the address space on ghost processes (G). Casper then intercepts all RMA operations issued to the user processes on this window and redirects them to the ghost processes. Since Casper does not migrate or copy the memory regions but just maps them into the ghost processes' address space, it benefits the RMA operations that are handled in MPI software stack without changing the behavior of any RMA operations already handled in hardware. This implementation can be summarized in the following steps.

a) Deployment of Ghost Processes: When the application is launched, Casper keeps aside a user-defined (through an environment variable) number of ghost processes. This strategy is implemented by generating at the MPI initialization stage a subcommunicator including only user processes and then transparently replacing **COMM_USER_WORLD** with this subcommunicator in all MPI calls through PMPI redirection. Then the ghost processes simply wait to receive any commands from user processes in an **MPI_Recv** loop during the whole execution in order to force the MPI implementation to make progress on any RMA operations that are targeted to those ghost processes (see Figure 4(b)).

b) RMA Memory Allocation and Setup: When the application creates an RMA window by using the **MPI_Win_allocate** function, Casper internally allocates a shared-memory region for each user process and its ghost processes (using MPI-3 **MPI_Win_allocate_shared** calls) in order to allow remote access from ghost processes to the user window regions that are located in the memories of the user processes. Then Casper creates a new window with the same memory regions that contains only the user processes, and returns it to the application.

c) RMA Operation Redirection: Once the user windows are shared with ghost processes, Casper transparently redirects, through PMPI redirection, the communication synchronization calls (i.e., lock, fence, post-start-complete-wait) and RMA operations to the ghost processes on the target node.

When multiple ghost processes are available on the target node, Casper spreads communication operations across them. This approach allows the software processing required for these operations to be divided among the different ghost processes, thus improving performance.

We believe Casper is a more suitable approach than other traditional attempts for GA-based applications on modern multicore and many-core architectures. Casper allows users to flexibly control the number of cores dedicated to asynchronous communication and computation, thus minimizing the performance impact to GA-based applications, which are often dominated by intensive computation that relies heavily on the computational resources. Furthermore, since Casper does not require multithreaded safety or system interrupts, the overhead resulting from asynchronous progress is greatly reduced.

VI. Evaluation

In this section we first describe the experimental platform and the software used. We then describe the experimental scenario and compare our solution with two thread-based approaches in both strong-scaling and weak-scaling forms.

A. Experimental Environment

Our experiments are executed on the NERSC Edison Cray XC30 supercomputer. Each node of Edison has 64 GB of DDR3 memory and two Intel® Xeon® E5-2695 v2 processors. The sockets are populated with running at 2.4 GHz, yielding a 19.2 GFLOPS/core performance. Edison employs a Cray Aries interconnect, and its nodes are disposed following a Dragonfly topology, leveraging up to 9.7 GB/s unidirectional MPI bandwidth and 1.3 μs internode MPI latency.

We use NWChem version 6.3 on top of Cray MPI 6.3.1 as the underlying MPI implementation. On its "regular mode" Cray MPI executes all RMA operations in software with asynchronous progress possible through background threads (by setting the **MPICH_ASYNC_PROGRESS** environment variable). Cray MPI can be also executed in a "DMAPP-based mode" that executes contiguous put and get operations in hardware—although still executing accumulate and noncontiguous operations in software—leveraging interrupt-based

TABLE I
CORE DEPLOYMENT IN NWCHEM EVALUATION ON CRAY XC30

Approach	Computing Cores	Async. Cores
Original MPI	24	0
Casper	23	1
Thread (O)	24	24
Thread (D)	12	12

asynchronous progress. As discussed in Section IV, GA applications have difficulty benefiting from an interrupt-based approach because of the large overhead brought by frequent interrupts. Therefore, in this paper we focus on the *regular mode*.

B. Scenarios

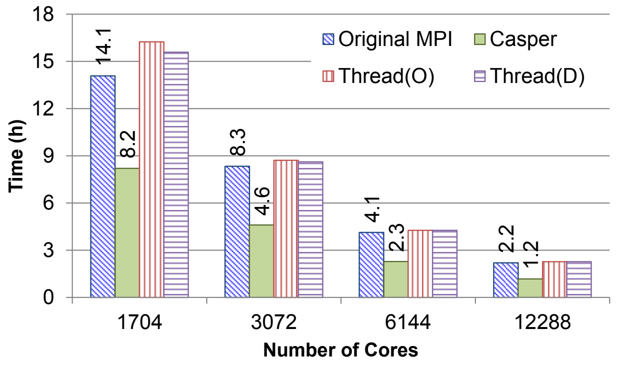
We evaluated the improvement of NWChem by comparing Casper with two thread-based approaches. The first approach—Thread (O)—employs oversubscribed cores where every thread and its associated MPI process execute on the same core. The second approach—Thread (D)—deploys dedicated cores where threads and MPI processes are executed on separate cores. We use the same total number of cores in all approaches, some of which are dedicated to asynchronous ghost processes/threads as listed in Table I. To compare the computational efficiency, we use the T_{comp} measured for the original NWChem as the base and use the number of cores including both the computing and the asynchronous cores employed in each experiment as N. We focus on the (T) portion of the CCSD(T) method and measure different water molecule $(H_2O)_n$ problems—denoted W_n for short—with double-zeta basis sets (cc-pVDZ from the NWChem basis set library).

C. Results

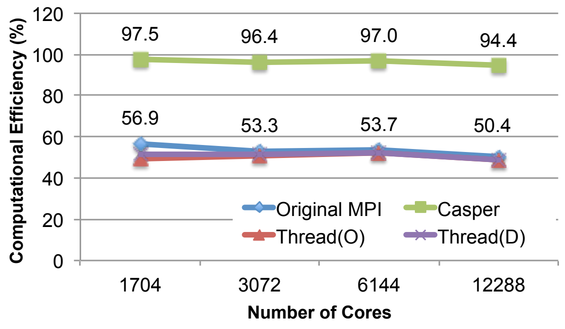
We first focus on the strong-scaling performance in large W_{21} (with pVDZ) problems by using a varying number of cores. Figures 5(a) and 5(b) show the execution time and computational efficiency, respectively. Relative to the original version, Casper is almost twice as fast, attaining close to 100% computational efficiency, whereas the thread-based approaches cannot improve the efficiency and perform even worse than the original implementation.

To determine the reason for these results, we measured separately the time consumed by the internal MMM computation and that of the RMA communication. As shown in Table II, although both Casper and the thread-based approaches eliminate the delay in RMA communication with asynchronous progress, the performance of the computation is negatively impacted. Since Casper spends only one core as a ghost process on each node, this degradation is reduced. With the thread-based approaches, however, performance is twice as bad because of oversubscribed cores or appropriation of half of the computing cores.

Figures 6(a) and 6(b) demonstrate the weak-scaling performance of the (T) portion with different problem sizes and core counts listed in Table III. Similar to the trend we observed in the first experiment, Casper doubles the performance and delivers close to 100% computational efficiency in all problem sizes, while the thread-based approaches impose a performance penalty with respect to the original MPI for all problem sizes. Hence, Casper enables large-scale executions at much more appealing run times.



(a) Execution Time.



(b) Strong-scaling computational efficiency.

Fig. 5. (T) portion in CCSD(T) for W_{21} with pVDZ.

TABLE II
(T) PORTION PROFILING FOR W_{21} WITH PVDZ

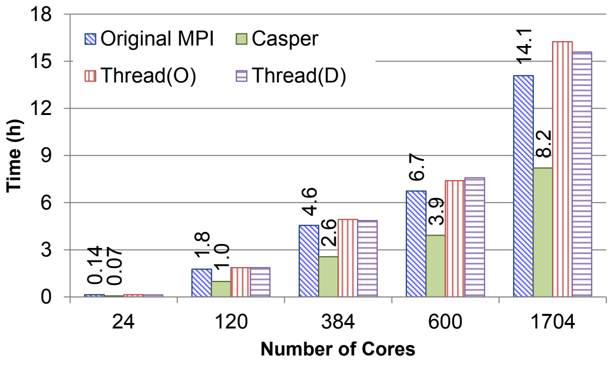
Approach	1714		3072		6144	
	MMM (h)	RMA (h)	MMM	RMA	MMM	RMA
Original	8.01	6.62	4.46	3.83	2.10	1.94
Casper	8.15	0.03	4.52	0.03	2.24	0.01
Thread(O)	15.49	0.12	8.57	0.08	4.23	0.04
Thread(D)	15.75	0.04	8.53	0.04	4.20	0.02

VII. CONCLUSION AND FUTURE WORK

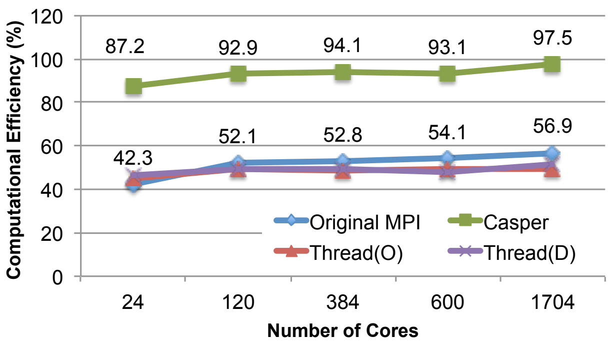
Chemistry computational applications, such as NWChem, often require large memory and disk space that exceed the capacity of a single node. The Global Arrays programming model is designed to solve this issue. It allows the user to build

TABLE III
CASPER RESOURCE USAGE IN NWCHEM EVALUATION ON CRAY XC30

Resource	W_5	W_{10}	W_{14}	W_{16}	W_{21}
Nodes	1	5	16	25	71
Cores	24	120	384	600	1,704



(a) Execution Time.



(b) Weak-scaling computational efficiency.

Fig. 6. (T) portion in CCSD(T) for varying W_n with pVDZ.

global dense arrays that are distributed across the memories of multiple nodes. MPI one-sided communication is the approach used to implement portable GA on various platforms.

In this paper, we have analyzed the gold standard CCSD(T) method in NWChem, one of the most important CC simulations implemented with highly efficient algorithms. We have observed close to 100% parallel efficiency on large water problems, however, with only 50% computational efficiency in the traditional approaches. Such scaling bottlenecks are caused by the lack of fully asynchronous communication capabilities in most MPI implementations. That is, the target process still has to make MPI progress to complete any one-sided operations issued on it as a target. This lack of asynchronous progress severely restricts the performance and scalability of GA-based applications, which often perform intensive computation interleaved with numerous one-sided operations.

We have utilized "Casper," an efficient process-based asynchronous progress model for MPI RMA communication to scale the computation on water clusters. Unlike traditional thread-based or interrupt-based asynchronous progress approaches, Casper provides low-overhead asynchronous progress without overuse of computational resources, an approach that is more suitable for large-scale GA-based applications. With Casper, we have scaled the water molecule problems at close to 100% computational efficiency on up to 12288 cores. We have also shown that for a very large water problem W_{21}, we can reduce the execution time around 50%.

After addressing the most time-consuming part of the NWChem analysis, in future work we plan to address the combined performance of the CCSD and (T) portions: featuring widely different communication-computation workloads, their optimal number of ghost processes differs. So far, however, Casper does not incorporate a dynamic solution.

Besides CCSD(T), we plan to study other modules of NWChem, which may have different performance characteristics. Specifically, as our next step we will focus on the self-consistent field module [15], another important module in NWChem.

Acknowledgments

This material is based upon work supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357. The experimental resource for this research was provided by the National Energy Research Scientific Computing Center (NERSC) on the Edison Cray XC30 computer.

References

[1] E. J. Bylaska et. al., "NWChem, a computational chemistry package for parallel computers, version 6.3," 2013.

[2] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A portable "shared-memory" programming model for distributed memory computers," in *ACM/IEEE conference on Supercomputing*, 1994.

[3] C. Oehmen and J. Nieplocha, "ScalaBLAST: a scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 8, pp. 740–749, 2006.

[4] H. E. Trease, "Code development for NWGrid/NWPhys," *Laboratory Directed Research and Development Annual Report-Fiscal Year 2000*, p. 103, 2001.

[5] J. S. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the global arrays PGAS model using MPI one-sided communication," in *Parallel and Distributed Processing (IPDPS)*, 2012.

[6] M. Si, A. J. Peña, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An asynchronous progress model for MPI RMA on many-core architectures," in *Parallel and Distributed Processing (IPDPS)*, 2015.

[7] Argonne National Laboratory, "MPICH — high-performance portable MPI," http://www.mpich.org, 2015.

[8] Cray Inc., "Cray Message Passing Toolkit," http://docs.cray.com/books/S-3689-24, Cray Inc., Tech. Rep., 2004.

[9] The Ohio State University, "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE," http://mvapich.cse.ohio-state.edu, 2015.

[10] Intel Corporation, "Intel MPI library," http://software.intel.com/en-us/intel-mpi-library, 2015.

[11] W. Gropp and R. Thakur, "Thread-safety in an MPI implementation: Requirements and analysis," *Parallel Comput.*, vol. 33, no. 9, pp. 595–604, Sep. 2007.

[12] M. Gilge, *IBM System Blue Gene Solution: Blue Gene/P Application Development*. IBM, Jun. 2013.

[13] S. Kumar and M. Blocksome, "Scalable MPI-3.0 RMA on the Blue Gene/Q supercomputer," in *EuroMPI/Asia*, 2014.

[14] J. R. Hammond, S. Krishnamoorthy, S. Shende, N. A. Romero, and A. D. Malony, "Performance characterization of global address space applications: A case study with NWChem," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 2, pp. 135–154, 2012.

[15] J. L. Tilson, M. Minkoff, A. F. Wagner, R. Shepard, P. Sutton, R. J. Harrison, R. A. Kendall, and A. T. Wong, "High-performance computational chemistry: Hartree-Fock electronic structure calculations on massively parallel processors," *International Journal of High Performance Computing Applications*, vol. 13, no. 4, pp. 291–302, 1999. [Online]. Available: http://hpc.sagepub.com/content/13/4/291.abstract