JCTC Journal of Chemical Theory and Computation

Designing a Scalable Fault Tolerance Model for High Performance Computational Chemistry: A Case Study with Coupled Cluster Perturbative Triples

Hubertus J. J. van Dam,* Abhinav Vishnu,* and Wibe A. de Jong*

Pacific Northwest National Laboratory, 902 Battelle Boulevard, Richland, Washington 99354-1793, United States

Received August 6, 2010

Abstract: In the past couple of decades, the massive computational power provided by the most modern supercomputers has resulted in simulation of higher-order computational chemistry methods, previously considered intractable. As the system sizes continue to increase, the computational chemistry domain continues to escalate this trend using parallel computing with programming models such as Message Passing Interface (MPI) and Partitioned Global Address Space (PGAS) programming models such as Global Arrays. The ever increasing scale of these supercomputers comes at a cost of reduced Mean Time Between Failures (MTBF), currently on the order of days and projected to be on the order of hours for upcoming extreme scale systems. While traditional disk-based check pointing methods are ubiquitous for storing intermediate solutions, they suffer from high overhead of writing and recovering from checkpoints. In practice, checkpointing itself often brings the system down. Clearly, methods beyond checkpointing are imperative to handling the aggravating issue of reducing MTBF. In this paper, we address this challenge by designing and implementing an efficient fault tolerant version of the Coupled Cluster (CC) method with NWChem, using in-memory data redundancy. We present the challenges associated with our design, including an efficient data storage model, maintenance of at least one consistent data copy, and the recovery process. Our performance evaluation without faults shows that the current design exhibits a small overhead. In the presence of a simulated fault, the proposed design incurs negligible overhead in comparison to the state of the art implementation without faults.

1. Introduction

In the past couple of decades, computational chemistry has developed to the point where it has become a significant tool in solving real world problems. The development has been supported by a tremendous growth in computational power provided by the most modern supercomputers. While this growth initially was delivered by increasing single processor capability, the past two decades have observed a significant part of this growth through the development of parallel computing systems. This development has recently been accelerated by the arrival of multicore and GPU-based systems. Currently, the extreme end of this development is embodied in the Jaguar machine,[1] which employs 224 256 cores, delivering at most 2.3×10^{15} floating point operations per second.[2] Presently, researchers are looking into the possibility of 1000 times more powerful supercomputers.[3]

At the same time, there has been a development toward parallel implementations of computational chemistry applications to take advantage of these supercomputers. A number of parallel quantum chemistry applications have become available, including NWChem,[4] GAMESS,[5] GAMESS-UK,[6] MOLPRO,[7] MOLCAS,[8] Q-Chem,[9] PQS,[10] MPQC,[11] ADF,[12] Dalton,[13] FreeON,[14] and COLUMBUS,[15] to name a few. The different applications and different methods

* Corresponding authors. Phone: +1 509 372 6441. Fax: +123 (0)123 4445557. E-mail: Hubertus.vanDam@pnl.gov (H.J.J.v.D.); Abhinav.Vishnu@pnl.gov (A.V.); Wibe.deJong@pnl.gov (W.A.d.J.).

10.1021/ct100439u
Published on Web 12/08/2010

implemented by these applications show different characteristics in the parallelism they can exploit. Ongoing developments continue to improve these capabilities, and chemistry continues to be at the forefront of parallel applications, as recently demonstrated by Apra et al.[16]

However, the increasing scale of these supercomputers has reduced the Mean Time Between Failures (MTBF) sharply. The current generation of supercomputers exhibits an MTBF of 6 h,[17] seriously limiting the overall system usage of these supercomputers. To deal with such problems, disk-based checkpoint/restart fault recovery strategies have usually been implemented. In this approach, the state of an application is written to disk at regular intervals. After a fault occurs, the last valid state is restored and the calculation continues from that point. This mechanism can be provided by the application itself writing restart files at particular points during a calculation or by the operating system saving the state of the calculation in an application-transparent manner. The problem with the checkpoint/restart approach is that the I/O overhead it introduces is significant. In fact, it is estimated that, somewhere between peta- and exa-scale calculations, the time spent on the overhead of the checkpoint/restart fault tolerance strategy will exceed the time spent doing useful computation.[18] In addition, quite a few times, the act of checkpointing itself brings the overall system to a halt. Clearly, other fault tolerance strategies for efficient recovery are imperative for computational chemistry methods. This leads to the following challenges:

• designing fault tolerant methods that allow computational chemistry codes to proceed correctly in the presence of faults
• ensuring that the overheads introduced are insignificant

These challenges were considered by Nielsen et al.[19] to some extent, but they judged implementing a fault tolerant method impractical for the lack of fault tolerance support in MPI. In this paper, we address these challenges by designing a fault tolerant strategy based on in-memory data redundancy using Global Arrays (GA).[20] GA uses a message passing layer, but it can use its own TCGMSG[21] layer rather than MPI. Although we have no control over MPI, we can design a fault tolerant GA/TCGMSG infrastructure as the foundation for fault tolerant applications. The aim of this approach is to develop applications that can withstand system failures while incurring low overheads. While the approach is not specific to chemistry applications, the impact is related to algorithm-specific parameters such as data volumes, message frequencies and sizes, communication/work balance, etc. Therefore, it is relevant to assess the appropriateness of the chosen approach to the target applications. To this end, we have implemented a fault tolerant version of the coupled cluster perturbative triples. We assess the level of fault tolerance that can be attained, the extent of the code modifications needed, as well as the overheads incurred.

The rest of the article is organized as follows. In section 2, we present the background of our work. In section 3, we present the design of our work. Section 4 presents the performance evaluation of the fault tolerant coupled cluster method. In section 5, we present the conclusions and future directions. We begin with the description of the background.

2. Background

The increasing scale of modern supercomputers has not only reduced the MTBF, it has also increased the variety of faults which occur during the execution of the applications. While disk failures continue to decrease due to the arrival of diskless systems (Cray XT5 and IBM Bluegene), processor failures due to overheating, node failures due to power tripping, and network failures for high connectivity interconnects are ubiquitous. Some of these failures lead to a permanent change of the calculation, e.g., a node failure due to power tripping leads to the permanent loss of the processes running on that node. These kinds of failures are referred to as hard failures.

Other failures, such as certain kinds of random memory bit flips as a result of radiation,[17] are transient or intermittent. They do not occur at a specific time and may not be reproducible at a later stage. These kinds of failures are referred to as soft failures. They are also inherently harder to detect because if such a fault is suspected there is no general *a posteriori* test that can reliably establish that such a failure actually has happened. The only way to reliably test for such faults is through redundancy, either in hardware, e.g., extra bits used in Error Correction Code (ECC),[22] or software by redundant execution of instructions and checking that the results are the same.[23]

During most extreme scale application executions, combinations of these failures are expected to occur. In the present work, we target only hard failures. Moreover, we assume that most hard failures will be detected by the runtime system and result in the affected processes being terminated. In particular, we assume that in most cases the impact will be such that all processes on a particular node will be terminated. Obviously this does not mean that soft failures are insignificant. However, provided the capability to detect soft failures is available, it is always possible to render soft failures into hard failures by terminating the affected processes.

2.1. Fault Behavior and General Data Fault Resilience. Given that a fault expresses itself in the termination of one or more processes, there are generally three different possible choices. The most commonly supported choice at the moment is to restart the application on the basis of a checkpoint/restart mechanism. As mentioned in the Introduction, the overhead of this approach is high and expected to become impractical for the scale of calculation currently envisioned.

Another approach is to detect the process failures and recover by reinstantiating the failed processes. In the case of processes failing due to hardware becoming unavailable, e.g., a power supply failing, effectively turning a number of processors off, the processes would have to be reinstantiated on additional resources. These resources would have to be on stand-by for as long as no fault occurs, which is wasteful. However, more problematic is that upon reinstantiating the processes they would have to be put in a state that is consistent with the other processes in the calculation. For many chemistry applications, this is highly nontrivial, as in normal circumstances, the state is updated in the course of the calculation. Hence, the state is a function of the program and the path its execution has taken through it up to a

particular point. This path often involves the modification of many variables that manage the calculation to be executed, such as I/O, communication, data distributions, etc. The application relies on most of these variables at a later point in the calculation. Hardly ever is the relationship between the values of these variables and the particular phase of the computation in hand explicitly formulated. Therefore, recreating a process and bringing it to the correct state is at best hard to achieve.

The remaining approach is to accept the loss of the failed processes and continue the calculation without them. Obviously, this leads to a degradation of performance, but as long as this is roughly proportional to the fraction of processes lost, this is acceptable. In most cases, the loss of performance has less of an impact than the interruption caused by checkpoint/restarting or forcing an application shutdown. In extreme cases where the loss of performance is severe, this essentially points to a serious instability of the machine, and continued execution is not recommended anyway.

Considering the implications of the various approaches, we consider the last option of continued execution in the degraded mode the most promising option. In designing such an approach, an important question is how to handle the impact a fault has on the data in the application. Clearly, as a fault leads to the loss of processes, the data those processes hold become inaccessible to the application. Broadly speaking, there are two possible ways to address this issue (disregarding keeping copies on disk, which would effectively be equivalent to checkpoint/restart).

One way to address this is to replace the lost data with "reasonable" guesses for the lost data. If the guesses are close enough and the amount of data lost small enough, one could expect the calculation to continue almost as normal with only a slight perturbation. This approach could work for optimization algorithms where the end-point is specified in terms of certain conditions being met. In this case, a small upset along the way might not have significant consequences. The main flaw in this idea is however that many data structures have to satisfy specific conditions. For example, molecular orbitals have to be orthonormal, the trace of the density matrix has to equal the number of electrons, wave functions have to be normalized, etc. Just replacing a patch of the data with some guess is more than likely going to violate these conditions. Explicitly re-establishing these conditions is often expensive and involves a global response; think for example of reorthonormalizing the molecular orbitals. The alternative would be to write the algorithm in such a way that it intrinsically strives to satisfy these conditions; i.e., meeting these conditions to a certain degree becomes part of the convergence criterion. In the abstract, it is not clear what the computational cost implications of such an approach are, but it is clear that this would require nontrivial modifications of all algorithms to which it is applied. Even then, this approach could only be used for optimization algorithms. It offers no way forward, for example, for algorithms like the MP2 energy evaluation, which is a single step method. In those cases, any perturbed input data will lead to a perturbed answer.

The other approach is to rely on redundant data of some kind. Obviously, disk-based data redundancy is out of question due to the I/O overheads, but in-memory redundancy is an option. In particular, in quantum chemistry where the computational complexity is always of a higher order than the data storage requirements (only in the large systems limit for linear scaling methods both orders become the same), it is possible to duplicate the critical data. The assumption is that the choice of the number of processors is motivated by the amount of work to be done rather than the amount of memory needed. This is certainly true for correlated quantum chemistry methods. The main cost of this approach will be in the extra communication required to maintain the duplicate copies. Provided the granularity of the work is high enough, this should not be a major problem.

In this paper, we describe how fault tolerance can be achieved using in-memory data redundancy in the context of degraded mode continued execution.

2.2. Global Address Space Programming Models and Global Arrays. As will become clear in the next section, a critical capability for our fault tolerance approach is that other processes can continue from the point where some process failed. Central to this capability is access to the data required to do so. Although in principle this access can be arranged in a number of ways and environments, in practice it is particularly facilitated by the Partitioned Global Address Space paradigm (PGAS). In this paradigm, physically distributed objects can be created, but the access mechanisms make it appear as if they exist in a shared memory space. This also means that every process can access all of each such object independently of what other processes are doing. Today, this programming model is supported by a number of programming languages such as Unified Parallel C[24] and Co-Array Fortran.[25]

However, before these programming languages were defined, it was already realized that the PGAS paradigm can be very helpful in solving some problems. In particular, in the context of the development of NWChem, this paradigm offered a way to deal with collections of tasks with a large spread in execution time without incurring large load imbalance overheads. To support this paradigm, the Global Array toolkit (GA) was created.[20] This library provides an efficient and portable "shared-memory" programming interface for distributed-memory computers. Each process in a Multiple Instruction Multiple Data (MIMD) parallel program can asynchronously access logical blocks of physically distributed dense multidimensional arrays, without the need for explicit co-operation by other processes. Unlike other shared-memory environments, the GA model exposes to the programmer the nonuniform memory access (NUMA) characteristics of high-performance computers and acknowledges that access to a remote portion of the shared data is slower than to the local portion. The locality information for the shared data is available and can be exploited to maximize the performance of codes if desired.[20]

Combining the global data access characteristics provided by the GA with data redundancy, it becomes particularly easy to ensure access to all data even in the case where processes fail. Hence, this library provides a highly suitable

platform with which to construct fault tolerant applications. Obviously, when the GAs were first written, fault tolerance was not envisioned as a problem. Therefore, changes to the library itself are required to make it fault-resilient. These changes are nontrivial, as they touch every layer in the library's software stack. They include considering how the library should respond to basic communication calls involving failed processes, implementing collectives such as barriers with incomplete sets of processes, as well as fault detection mechanisms to establish which processes have been lost and propagating that information to the processes that need to know, and modifying the process manager (the equivalent of mpirun for MPI applications) to respond correctly to the loss of processes. This work is currently ongoing, but its discussion is beyond the scope of this paper. Instead, in this paper, we limit ourselves to describing how a fault-resilient GA implementation can be used to build fault tolerant computational chemistry applications. For performance evaluations, a traditional GA library is used, and faults are simulated by making selected processes behave as if they have failed. This triggers the same responses in the application and the fault tolerance layer as a real fault. Therefore, it is expected to give a realistic impression of the behavior of the real fault-tolerant code.

2.3. The Coupled Cluster Method. The Coupled Cluster (CC) method has since its inception[26] developed into the dominant method for highly accurate calculations of total energies. Due to the relatively high cost of the method, a hierarchy of methods has been developed of increasing accuracy and cost. The natural relationship between CC and perturbation theory allows for the addition of perturbative corrections that offer increased accuracy at a lower cost than the full CC method at the corresponding level of approximation. The most widely used method of this kind is the CCSD(T) method. This refers to a CC method in which the singly and doubly excited Slater determinants are accounted for in the iterative solution of the equations, the CCSD part. The triply excited Slater determinants are accounted for using perturbation theory, which is indicated with (T).

The theory of CC methods has been described in a number of papers.[26−29] In this paper, we concentrate on creating a fault-tolerant implementation of the approach by Kobayashi and Rendell.[30] The initial implementation is characterized by an iterative CCSD solver that has a cost that scales as $O(N^6)$. Every iteration, this solver writes out the current values of the amplitudes, which allows this part to be restarted. The CCSD part is followed by the (T) term to form CCSD(T). This part requires transformed integrals with at most three virtual indices, which cost $O(N^5)$ to compute, followed by the triples contributions that cost $O(N^7)$. Because the triples contributions are calculated perturbatively, they are evaluated in a single pass. Although at $O(N^7)$ the triples contributions are the most expensive part because they are evaluated in a single pass, there is no natural point from which to restart their evaluation.

Despite the algorithm having no natural restarting point, this does not mean that it is impossible to make the code restartable. In doing so, the transformed two-electron integrals would have to be recalculated, as storing them would require excessive amounts of disk space. For the triples contributions themselves, the total number of tasks could be divided into blocks of some fixed number of tasks. At the end of each block, the current total triples contribution could be saved along with the task number of the next task. This would provide the data needed for restarts. The disadvantages of doing this are increasing the load imbalance due to extra barriers needed at the end of every block of tasks and a loss of productivity, as restarting a calculation requires resubmitting the job and waiting for its execution. Hence, having a facility that allows the code to continue without restarting is usually beneficial.

In addition to the fact that it is beneficial not to have to restart the perturbative triples, their high cost and good scalability make this code one of the prime candidates to be executed on extreme scale computing platforms. Indeed, this code was run on 200 000+ cores on Jaguar already.[16] So this code is likely to run in cases where fault tolerance is essential and therefore a suitable target application for our approach. In the future, we plan to extend our fault tolerance approach to other code modules as well.

3. Overall Design

In the following sections, the model for achieving fault tolerance based on in-memory redundancy is outlined. Important questions pertaining to this model are

- What kind of failure scenarios can this model address?
- How extensive are the code changes needed to integrate this model into an application?
- What are the performance overheads if no faults occur?
- What is the overhead to recover from a fault? To answer these questions, we built a fault-tolerant version of CCSD(T) using the model described here. With this implementation, we illustrate the answers to these questions.

A general aspect in a redundancy-based fault tolerance model is that when a process fails its memory becomes inaccessible, and the data it held are lost. Therefore, these data need to subsequently be fetched from a different location. To facilitate these changes in the communication pattern, it is essential to have a memory management approach that is sufficiently flexible. The GA[31] provides a virtual shared memory programming model that gives every process the same view of a distributed data structure. This way, it offers sufficient flexibility for the approach we propose. Therefore, on the application side, we have built an infrastructure on top of the GA that manages the application response to faults, as outlined below. We have currently integrated this infrastructure with NWChem[32] to deploy it in real chemistry applications.

NWChem breaks large-scale calculations up into small tasks that work on particular blocks of data. When a failure occurs, the responsibility for a particular task needs to be transferred to another process. To enable this, visibility of a particular process's work to other processes has to be ensured.

Combining the knowledge of a process's state and the redundancy of the data, it is possible to devise a recovery mechanism that returns the overall calculation to a valid state after a fault has occurred. In the following subsections, we

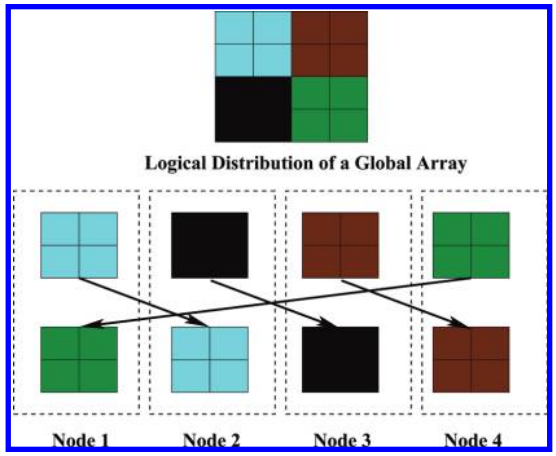

Figure 1. Relationships between the logical array, the physically distributed primary copy, and the distributed shadow copy.

present the details of the data storage model, the task model, and the fault recovery mechanism. This description of the general approach is followed by a description of how this is applied in the triples part of the CCSD(T) code.

3.1. The Data Storage Model. As stated in the outline of the general approach, data redundancy is used to enable fault recovery. This means that for every distributed data object a primary copy is created that is identical to a GA as usual. Figure 1 shows, at the top, the logical view of the GA. This array is partitioned into blocks across a processor grid. The choice of the processor grid depends on the number of dimensions in the GA, the sizes of the dimensions, and the number of processors. In practice, a single process is associated to a given "processor" in the processor grid. After this partitioning, every process is given at most a single data block of the GA, as shown in the middle.

In addition, a shadow copy is created. The data in the shadow copy are shifted by a given number of processes relative to the primary copy in a cyclic fashion, as shown at the bottom. The shift is referred to as the "processor shift" and can be chosen arbitrarily as long as it is not an integer multiple of the number of processes (that would lead to both the primary and shadow copies of the data residing at the same process). Obviously, under the assumption that all processes on a single node are likely to fail together, the processor shift should be at least equal to the number of processes per node to ensure that the shadow copy resides on a different node than the primary copy.

The processor shift is implemented using the new restricted array capabilities of the global arrays. This allows for physical data blocks to be mapped to processes using an arbitrary mapping. Hence, all of the administration related to the mapping between the primary and shadow copies can be off-loaded to the GA. In addition, the restricted array mappings also allow mappings that are not simply a cyclic shift. For a given machine, this capability can be exploited to place the data such that the application fault tolerance is maximized while minimizing the communication cost. Doing this requires knowledge of the anticipated fault behavior of the machine as well as the topology of the interconnect. At present, we do not exploit this, but in the future we could do so if that turns out be beneficial.

If being able to access data after a failure was the only concern, it would be sufficient to have only the primary and shadow copies. In practice, however, it is possible for a process to fail while it is updating data held on remote processes. This data will still be accessible after the failure but its contents might be corrupted. Therefore, a facility is needed to record the status of every data block. Hence, for both the primary and shadow copy, an additional GA is needed with a length of the number of processes. These arrays serve as flags to signal whether the corresponding data block at the particular process is "clean" or "dirty".

In summary, in this approach, every GA in the original implementation is replaced with a redundant GA for fault tolerance. Every redundant GA consists of four GAs, the primary copy, the shadow copy, the primary copy status flags, and the shadow copy status flags. In our implementation, this collection of data structures is treated as a single object with an interface that is essentially the same as the one for the usual GAs.

In analogy to the usual GA implementation, the interface to manage these data structures implements five routines:

(1) nsft_create: This creates an N-dimensional distributed array with its associated redundant copies and the block status flags.
(2) nsft_destroy: This clears a distributed array and its associated data structures up.
(3) nsft_map_put: This stores data in a distributed array, managing access to the shadow copies and the status flags. It can transfer data to multiple arrays in a single call.
(4) nsft_map_acc: This is the same as nsft_map_put apart from the fact that, whereas put overwrites the original data, the accumulate (named "acc" for short) routine adds to it.
(5) nsft_get: This gets a patch of data from any of the copies that are available.

The put and accumulate routines are for the most part straight equivalents of the corresponding routines in the GA library. However, the need to be able to update multiple arrays in one call is a requirement of the fault recovery, and the details of this will be discussed below.

3.2. The Task Model. Traditionally, the work in a given compute phase is broken up into tasks. Any task in some way involves

- establishing which task to do
- retrieving the data needed to complete the task
- executing some instructions to generate results
- storing the results in the appropriate places

In practice, these steps do not necessarily have to be executed in this order. For example, a task may get some data, do a bit of work, store some results, get some more data, do more work, and accumulate the result in a local buffer (the buffer could be globally summed after all tasks have been completed). However, if the calculation has to be able to recover from faults, a stricter set of rules is needed.

First of all, in order to have a simple way to identify what every process is doing, every task is associated with a unique number within a particular computational phase. This task number dictates what work is involved in the task, what data

it needs, and, most importantly, what data it is going to change. The advantage of this is that if a task needs to be re-executed due to a fault, the recovery mechanism only has to pass the task number to a suitable process to identify what needs to be done. To some extent, task numbers are already implicitly used in this way but often with the assumption that the task numbers are monotonically increasing. In a fault-tolerant application, this assumption no longer holds, as after a fault, the recovery mechanism may pass an old task number to a process for re-execution.

So as a general rule, the task model will insist that a process loops over tasks. The tasks may come in a random order, and whenever a task number is in the valid range from 1 to the number of tasks, the task parameters are calculated from the task number, and the required work is done. If the task number is outside the valid range, this means that all tasks have been done. Typically, the process then proceeds to a barrier that marks the end of the compute phase in question.

As far as accessing data is concerned, the fault-tolerant task model imposes extra rules only on the way data are changed. As a general rule, fault tolerance requires that when a fault occurs the state of a calculation can be either restored to the point where it is as if a task was never started or progressed to the point where it is as if a task was fully completed. This all or nothing approach is very much akin to transactional systems.[33] Transactional approaches are commonly used in distributed databases and business administration systems. They have two characteristics that are crucial to these application areas: atomicity and serializability. Atomicity in this context means that all modifications of all data items in a transaction are either progressed to completion or no updates happen at all, leaving all persistent data structures in their original state. Serializability means that all transactions complete as if they were executed in serial in some order. This implies that no process can get a view of the data where only part of the transaction has taken place, i.e., any of the intermediate states of the transaction. The implementation of these approaches rely on a two-stage mechanism for executing transactions. In the first phase, the data needed for the updates are sent out to all of the relevant data servers, but the persistent state is not changed. In a second phase, the changes are committed, meaning that the changed data structures take effect in the persistent state and the previous versions are discarded. If for some reason not all servers can commit, an abort can be issued, causing all modified versions to be discarded and the unmodified versions to remain part of the persistent state. Crucial is the separation between the data transmission and changing the persistent state of the data.

Although, in principle, elements of transactional updates could be implemented in the GA, at present such mechanisms are not available. The reasons for not supporting these mechanisms are related to both need and cost. If fault tolerance is not an issue, then there is no benefit in having a transactional mechanism. To the contrary, because on the data server side the information flows through an additional copy, three memory accesses are needed for each data element. In the current implementation, only one memory access is needed when the data are directly received into the persistent data set. In addition, the transactional approach requires nontrivial amounts of extra memory as well for storage of data between transmission and committing.

In our approach, the basic GA functionality is unchanged. This, however, requires that all primary copies are updated at the same time, so that if a fault occurs, all of those blocks can be marked dirty, and the unchanged shadow copy blocks count as the true record of the data. It also requires that all of the shadow copies be updated at the same time, so that if a fault occurs at that stage, all affected shadow copy blocks can be marked dirty, and the already updated primary copies count as the true record of the data. Obviously, a scenario in which some arrays have both their primary and shadow copies updated whereas others were left unchanged altogether leaves the calculation in an unrecoverable state. To facilitate writing code that adheres to these data update requirements, the map-based put and accumulate routines are provided. These routines can update as many arrays as needed in one call and ensure that the updates are executed in a way that allows for fault recovery. To ensure correctness, however, we also have to enforce that there can only be at most one call to update data per task. Finally, in order for a different process to continue from where a failed process left off, it is essential that every task stores its results in a distributed data array. No results shall be stored in local buffers beyond the scope of a task.

Obviously, for the fault recovery mechanism to decide on the right course of action, it needs to know what a process was doing at the time of failure. Of course, it needs to know which task was being executed, but it also needs to know what was being done on the task. In particular, we distinguish four different states:

(1) acquiring the task number
(2) working (getting data and computing results)
(3) updating primary data copies
(4) updating shadow data copies

These task states together with the task number are stored in a distributed and redundant GA with one element per process, so that the recovery mechanism can access this information when needed. The management of these states can be completely hidden from the programmer.

The task model needed for fault tolerance is summarized in Figure 2. The two gray sections are provided by the infrastructure for fault tolerance, whereas the white section is application-dependent code. The arrows indicate additional communication to record the state of a task in a globally accessible place. This way, the task state can be recovered in case the task fails.

3.3. The Fault Recovery Mechanism. The components discussed so far are essential ingredients in making fault tolerance possible. The fault recovery mechanism is the part that makes it work. It has to address the following questions:

- who executes the recovery mechanism?
- when is the recovery mechanism invoked?
- what steps are taken?

To answer the question of who is to respond to a fault, every process is assigned a "buddy" process. In this context, the two processes in a buddy pair are inequivalent in that

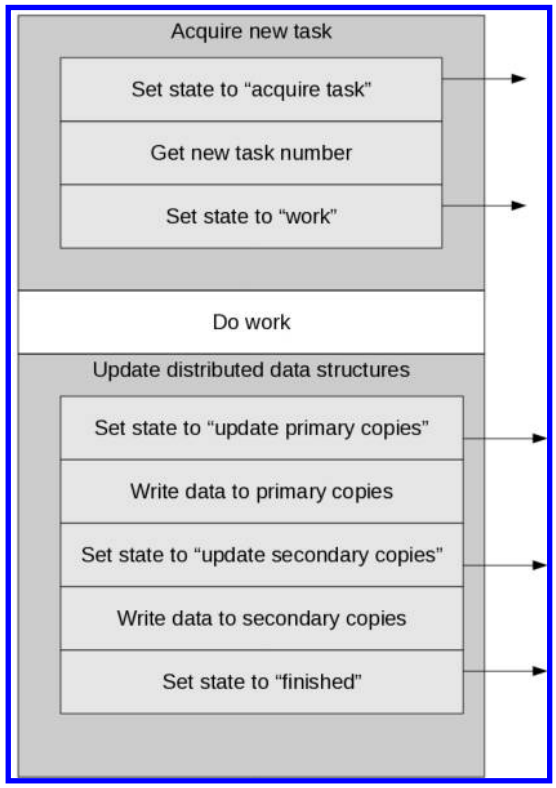

Figure 2. Outline of the stages in the fault-tolerant task model. The arrows indicate additional communication to record the task state changes needed.

the buddy of a process steps in and takes care of the process's responsibilities when it no longer can due to a fault. However, this relationship does not work the other way around. For a given process, its buddy is chosen by subtracting the processor shift (as used in the data storage scheme) of the process rank.

The most convenient time for the buddy process to respond is when it reaches its next time to acquire a new task number. This allows the routine that generates the new task number to take generic remedial steps, as outlined below, and if the failed task needs to be re-executed, its number can be passed to the buddy process as its next task. This way, the remedial steps can be almost completely hidden from the programmer, and no code has to be written to specifically deal with the re-execution of tasks. Instead, the normal code can be used for recovery-driven re-execution as well as normal tasks.

The recovery process follows a very simple approach to ensure that the data the program uses are in a consistent state and that, where necessary, tasks that failed are finished. The precise steps that need to be taken depend on the state of the tasks of the failed process at the time the fault occurred. Obviously, the buddy process first has to retrieve the task number and state of the failed process, which are stored in a redundant GA. If the state of the task was "acquiring new task", at present, recovery is not possible, and the calculation terminates. As shown in section 3.4, this situation is very unlikely to arise, but if it becomes a problem, it can be addressed by recording the status of every task rather than every process. If the state was "working", then no remedial action is needed, but the failed task needs to be re-executed. If the state was "updating primary copies", then the contents of the associated memory are undefined. Hence, the memory blocks associated with the failed task are worked out, and those blocks are marked "dirty". The programmer has to provide a routine to the task generator that can provide a list of blocks a task will change on the basis of the task number and process rank. As this relationship is highly dependent on the particular algorithm the program executes, there is no general other way of obtaining that information. If the primary copies were corrupted, the task results will not have been stored in the shadow copies either, and the task needs to be re-executed. Finally, if the state was "updating shadow copies" then the associated blocks of the shadow copies are marked "dirty". However, as the results are already stored in the primary copies, there is no need to re-execute the task in this case.

Concluding this section, it is clear that most of the details of the fault recovery can be hidden from the programmer. The only thing that cannot be hidden is the relationship between the task number and the data that task updates, as this is an intrinsic property of the application algorithm. However, generating this dependency should be trivial for most algorithms.

3.4. Summary of the Fault Tolerance Approach. Considering the approach outlined above, it is clear that this approach cannot deal with all faults. The approach explicitly assumes that faults lead to process termination and that soft faults do not occur. In addition, the failure of two processes differing in rank by the processor shift destroys both the primary and shadow copies of some of the data, leaving the calculation in an unrecoverable state. However, the likelihood of such failures is the square of the probability of any given process failing. That is assuming that the processor shift is chosen such that the likelihood of the one node failing is independent of a failure of the other. Hence, this should be very small for any reasonable machine.

Furthermore, at present, the approach cannot recover from a fault occurring when a process is acquiring the next task number. The reason is that if the global counter generated a task number but the receiving task failed before storing this number in a globally accessible place, then this task is lost. At present, this is not expected to be a significant problem, as the code spends very little time establishing the next task number compared to the rest that it does. So the chance of failing in this phase is very small. However, if needed there are at least two ways in which the problem can be addressed. One is to change the global counter to keep a record of the last task number it handed out to each process. Storing this record in a GA keeping all data elements on the same process as the global counter itself allows an efficient implementation. The reason is that the data are written often but read once. That is, the data only need to be read if a task failed while getting the next task number. Then, checking the consistency between the task state and the last global counter generated task number allows the determination of what task should be executed. The other way to address this issue is to store the state of every task. At the end of the compute phase, all tasks should be completed. Ones that are not must have remained uncompleted due to faults, and remedial actions can be triggered to complete them nevertheless.

Finally, the recovery process can cause a loss of data redundancy. If a process fails while updating the primary or shadow copies, the corresponding copy is marked "dirty", after which only the other copy remains. It is tempting at that stage to try and salvage the dirty copy with data from the clean copy. However, during the compute phase, this can lead to inconsistencies. To see this, consider processes A and B, which are executing tasks that are both going to update the same patch of a redundant GA. The primary copy of that patch resides on process P and the shadow copy on process S. Now process A fails while updating P, so the data are P is dirty. Process B has finished its task, updates P, and is about to update S. At this point, the salvage procedure kicks in copying the data from S to P and marking P "clean" again. Subsequently, process B updates S. At this time, the consistency between the redundant copies is lost. The primary copy contains data without the contribution from B, and the secondary copy includes the contribution from B. A similar scenario can cause problems attempting to salvage a "dirty" shadow copy by using the data from the primary copy. So these salvage operations should not be attempted during the compute phase when a number of processes could simultaneously be changing the redundant GAs. The only time such salvaging can be done is as part of the barrier at the end of a computational phase. Only at that time can one guarantee that copying data between the primary and shadow copies does not lead to inconsistencies due to missed updates.

The loss of redundancy due to dirty blocks is obviously a direct consequence from the way the GA implements data updates. If a transactional GA interface were implemented and used, this issue would not arise. Depending on how severe this issue becomes, this might be a reason to develop a transactional approach.

3.5. Code Changes Required for Fault Tolerance. Aside from the question of which faults this approach can deal with, there is the question about the extent of the code changes needed to implement it. In Figure 3, a section of pseudocode is shown that illustrates the changes needed to implement fault tolerance in the CCSD(T) code. The first obvious difference is how in the fault-tolerant version key task parameters are derived from the task counter. In the original version, they were established using an implicit dependency on the task counter, but as discussed that dependency is broken due to the fault tolerance. Second, fault tolerance requires that intermediate results are kept in a globally accessible space. Hence, the local variable "sum" had to be replaced by a redundant GA. The contributions to the energy are added on by the sft_map_acc routine, which applies consistent updates to the primary and shadow copies. In this case, it only adds to one array, but it can modify any number of arrays. Third, notice the additional parameter to sft_nxtask, which is a routine that informs the recovery process about the data blocks a given task number modifies. As this information is application-specific, this routine needs to be provided by the application programmer. Finally, after all contributions have been summed into "g_sum", the various partial results need to be collected and added together to arrive at the final answer. Obviously, some code changes are needed, and some thought needs to go into this to arrive at a fault-tolerant implementation. Nevertheless, the example clearly shows that these changes are relatively minor. In particular, the administration involved in maintaining a calculation in a recoverable state can be completely hidden from the application programmer.

```
itask = -1
sum = 0.0
next = nxtask()
do a = 1, nvirt
  do j = 1, nocc
    itask = itask + 1
    if (itask.eq.next) then
      call ga_get(g_in,1,nocc,
                  a,a,d,len)
      ... do work ...
      sum = sum + term
      next = nxtask()
    endif
  enddo
enddo
call ga_dgop(sum)
```

```
call sft_create(g_sum,1,nproc)
call sft_zero(g_sum)
numtasks = nvirt*nocc
next = sft_nxtask(access_map)
do while (next.le.numtasks)
  a = next/nocc+1
  j = next - (a-1)*nocc
  call sft_get(g_in,1,nocc,
               a,a,d,len)
  ... do work ...
  call sft_map_init(map)
  call sft_map_add(map,g_sum,
                   term)
  call sft_map_acc(map)
  next = sft_nxtask(access_map)
enddo
call ga_sync
sum = 0.0
do ip = 1, nproc
    call sft_get(g_sum,1,1,
                 ip,ip,term)
    sum = sum + term
enddo
call sft_destroy(g_sum)
```

Figure 3. Comparison of normal code versus fault-tolerant code.

4. Performance Evaluation

The maintenance of multiple copies of distributed data structures and the task states all come at a cost. Therefore, an important question is whether the approach suggested is actually practical or whether the overheads make it unusable. To assess these overheads, we performed calculations on uracil with a cc-PVTZ basis set[34] using Cartesian basis functions. The resulting calculation involves 340 orbitals, of which 29 are doubly occupied. All orbitals were correlated in the CCSD calculation. Subsequently, we performed CCSD(T) calculations both with the original code and with the fault-tolerant code.

The calculations were run on Chinook.[35] Chinook is a 160 TFLOP that consists of 2310 HP DL185 nodes with dual socket, 64-bit, Quad-core AMD 2.2 GHz Barcelona Processors. Each node has 32 GB of memory and 365 GB of local disk space. Communication between the nodes is performed using InfiniBand with Voltaire[36] switches and Mellanox[37] adapters. The system runs a version of Linux based on Red Hat Linux Advanced Server. A global 297 TB SFS file system is available to all of the nodes.

The calculations were run on three different processor counts, i.e., 256, 512, and 1024 processors. As the calculation is relatively small, it did not scale well to 2048 processors. For each processor count, the calculations were run three times, each time as a new job so that different node pools were used. Although this does not provide a statistically significant sample, it allows spurious results to be ruled out. The timings obtained were averaged over the three runs and

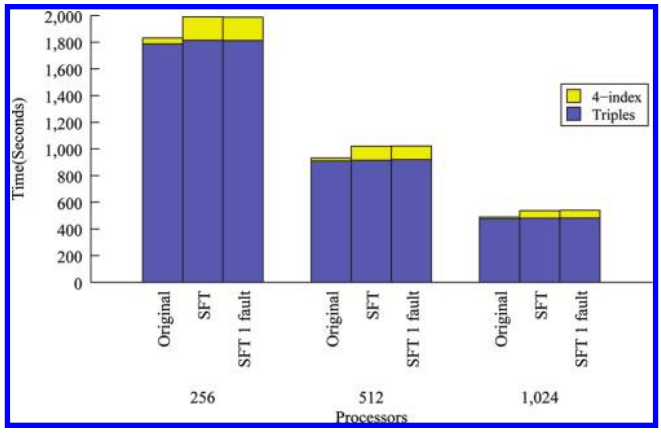

Figure 4. Performance evaluation, on 256, 512, and 1024 processors comparing the original code, the fault-tolerant code (SFT), and the fault-tolerant code with a single fault (SFT 1 fault). The total time is split into the four-index integral transformation and the triples energy contributions.

are displayed in Figure 4. The figure compares for each calculation the time taken by the original code, the fault-tolerant code without faults, and the fault-tolerant code with one fault at the beginning of the fault-tolerant code section. The results show that the overhead incurred is 8.7%, 9.5%, and 9.9% for 256, 512, and 1024 processors, respectively. This overhead is largely incurred during the integral transformation phase of the calculation; the triples evaluation incurs an overhead of only 1.5%. As the triples component has a complexity of $O(N^7)$ and the integral transformation $O(N^5)$, this problem becomes less severe for larger calculations. In addition, the overheads are relatively insensitive to the processor count. Hence, when no faults occur, the overheads are sufficiently low for this technology to be used in routine calculations. We also measured the fraction of the time spent obtaining new task numbers, for which we found that 0.13% of the time the code is in a state where it currently cannot recover from a fault. Hence, in the vast majority of fault instances, the code will be able to continue with the current implementation.

So far, the case where no faults occur was considered, but the overhead of recovering from a fault is very important for practical applications. However, establishing this overhead is slightly complicated by the fact that, by continuing in degraded performance mode, any measurements will mix two aspects. Initially, there is the cost of bringing the data back into a valid state plus potentially re-executing the failed task. After that, there is the impact of continuing the calculation with one less processor. In practice, the timings fluctuate too much to separate these two effects. Hence, we simulate a process failing on its first task, as this will have the maximum impact and thus provide an upper limit.

The results in Figure 4 show that the loss of a single process leads to a performance degradation of at most 0.71% where one processor corresponds to 0.39% of the total computing capacity with 256 processors. This performance degradation includes the initial execution of the failed task, the fault recovery, the re-execution of the failed task, and the loss of one processor's compute capacity for the remainder of the calculation. Overall, this performance degradation is on the same order of magnitude as running the calculation on one processor less throughout; this impact is definitely small compared to the impact of aborting the whole calculation.

5. Conclusions

In this paper, we have discussed the challenges posed to computational chemistry by the fact that extreme scale computers are likely to have relatively small MTBFs. To address this, we have considered making real computational chemistry code fault tolerant. The aim of this effort is to allow the program to correctly continue execution despite the loss of processes. We have presented an approach that offers this capability, integrated it into NWChem, and evaluated the performance of the method for the perturbative triples component of CCSD(T) calculations. It was shown that the fault-tolerant code runs with an overhead of only a few percent relative to the original code. The occurrence of a fault imposes an additional overhead of less than 1%. These overheads are acceptable in comparison to the alternative in which, currently, the effort expended on the calculation would be lost. Even in the ideal case where the triples would be restartable, the disruption caused by terminating and restarting the application would be significantly larger than the overhead of continuing the calculation in degraded mode.

We plan to deploy the infrastructure developed to other algorithms throughout NWChem. In particular, it is necessary to test the approach for algorithms where communication dominates more. The approach itself is not tied to NWChem, and we expect applications of it in other, most likely GA-based, codes as well.

Acknowledgment. This work was supported by the eXtreme Scale Computing Initiative at Pacific Northwest National Laboratory. Pacific Northwest National Laboratory is operated for the U.S. Department of Energy by Battelle. This work was done in part using EMSL, a national scientific user facility sponsored by the Department of Energy's Office of Biological and Environmental Research and located at Pacific Northwest National Laboratory, operated for the U.S. Department of Energy by Battelle under contract DE-AC05-76RL01830.

References

(1) Jaguar Cray XT5 supercomputer. http://www.nccs.gov/jaguar/ (accessed Mar 31, 2010).

(2) The TOP500 report. http://www.top500.org/lists/2009/11 (accessed Mar 31, 2010).

(3) Amarasinghe, S.; Campbell, D.; Carlson, W.; Chien, A.; Dally, W.; Elnohazy, E.; Hall, M.; Harrison, R.; Harrod, W.; Hill, K.; Hiller, J.; Karp, S.; Koelbel, C.; Koester, D.; Kogge, P.; Levesque, J.; Reed, D.; Sarkar, V.; Schreiber, R.; Richards, M.; Scarpelli, A.; Shalf, J.; Snavely, A.; Sterling, T. Exascale software study: Software challenges in extreme scale systems, 2009. The office of Advanced Scientific Computing Research. http://www.er.doe.gov/ASCR/Research/CS/DARPAexascale-software(2009).pdf (accessed Mar 31, 2010).

(4) Valiev, M.; Bylaska, E.; Wang, D.; Kowalski, K.; Govind, N.; Straatsma, T.; van Dam, H.; Nieplocha, J.; Apra, E.;

Windus, T.; de Jong, W. *Comput. Phys. Commun.* **2010**, *181*, 1477–1489.

(5) Gordon, M.; Schmidt, M. Advances in electronic structure theory: GAMESS a decade later. In *Theory and applications of computational chemistry*; Dykstra, C., Frenking, G., Kim, K., Scuseria, G., Eds.; Elsevier: New York, 2005.

(6) Guest, M.; Bush, I.; van Dam, H.; Sherwood, P.; Thomas, J.; van Lenthe, J.; Havenith, R.; Kendrick, J. *Mol. Phys.* **2005**, *103*, 719–747.

(7) Werner, H.-J.; Knowles, P. J.; Lindh, R.; Manby, F. R.; Schütz, M.; Celani, P.; Korona, T.; Mitrushenkov, A.; Rauhut, G.; Adler, T. B.; Amos, R. D.; Bernhardsson, A.; Berning, A.; Cooper, D. L.; Deegan, M. J. O.; Dobbyn, A. J.; Eckert, F.; Goll, E.; Hampel, C.; Hetzer, G.; Hrenar, T.; Knizia, G.; Köppl, C.; Liu, Y.; Lloyd, A. W.; Mata, R. A.; May, A. J.; McNicholas, S. J.; Meyer, W.; Mura, M. E.; Nicklass, A.; Palmieri, P.; Pflüger, K.; Pitzer, R.; Reiher, M.; Schumann, U.; Stoll, H.; Stone, A. J.; Tarroni, R.; Thorsteinsson, T.; Wang, M.; Wolf, A. *MOLPRO*, version 2009.1; University College Cardiff Consultants Ltd.: Cardiff, United Kingdom, 2009.

(8) Karlström, G.; Lindh, R.; Malmqvist, P.-A.; Roos, B.; Ryde, U.; Veryazov, V.; Widmark, P.-O.; Cossi, M.; Schimmelpfennig, B.; Neograd y, P.; Seijo, L. *Comput. Mater. Sci.* **2003**, *28*, 222–239.

(9) Shao, Y.; et al. *Phys. Chem. Chem. Phys.* **2006**, *8*, 3172–3191.

(10) Baker, J.; Wolinski, K.; Malagoli, M.; Kinghorn, D.; Wolinski, P.; Magyarfalvi, G.; Saebo, S.; Janowski, T.; Pulay, P. *J. Comput. Chem.* **2008**, *30*, 317–335.

(11) Janssen, C. L.; Nielsen, I. M. B. *Parallel computing in quantum chemistry*; CRC Press: Boca Raton, FL, 2008.

(12) te Velde, G.; Bickelhaupt, F. M.; Baerends, E. J.; Fonseca Guerra, C.; van Gisbergen, S. J. A.; Snijders, J. G.; Ziegler, T. *J. Comput. Chem.* **2001**, *22*, 931–967.

(13) *DALTON*, version 2.0; Chemistry Department at the University of Oslo: Oslo, Norway, 2005.

(14) Weber, V.; Challacombe, M. *J. Chem. Phys.* **2006**, *125*, 104110.

(15) Lischka, H.; Shepard, R.; Pitzer, R. M.; Shavitt, I.; Dallos, M.; T., M.; Szalay, P. G.; Seth, M.; Kedziora, G. S.; Yabushita, S.; Zhang, Z. *Phys. Chem. Chem. Phys.* **2001**, *3*, 664–673.

(16) Aprà, E.; Rendell, A. P.; Harrison, R. J.; Tipparaju, V.; deJong, W. A.; Xantheas, S. S. Liquid water: obtaining the right answer for the right reasons. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*; ACM: New York, 2009; pp 1−7.

(17) DeBardeleben, N.; Laros, J.; Daly, J.; Scott, S.; Engelmann, C.; Harrod, B. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development, 2010. Los Alamos National Laboratory Web Site. http://institute.lanl.gov/resilience/docs/HECResilienceWhitePaperJan2010final.pdf (accessed Mar 31, 2010).

(18) Daly, J. T. Application resilience for truculent systems. Presented at the 2009 Fault Tolerance Workshop for Extreme Scale Computing [Online], Albuquerque, NM, March 19−20, 2009. Teragrid Forum. http://www.teragridforum.org/mediawiki/images/8/80/Daly2009ws.pdf (accessed Mar 31, 2010).

(19) Nielsen, I. M. B.; Janssen, C. L.; Leininger, M. Scalable fault tolerant algorithms for linear-scaling coupled-cluster electronic structure methods, 2004. Sandia National Laboratories. http://prod.sandia.gov/techlib/access-control.cgi/2004/045462.pdf (accessed Sep 27, 2010).

(20) Nieplocha, J.; Harrison, R. J.; Littlefield, R. J. Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers. In *Proc. Supercomputing'94*; IEEE CS Press: Washington, DC, 1994; pp 340−349.

(21) Harrison, R. *Int. J. Quantum Chem.* **1991**, *40*, 847–863.

(22) Lin, S.; Costello, D. J., Jr. *Error Control Coding: Fundamentals and Applications*, 2nd ed.; Prentice Hall: Englewood Cliffs, NJ, 2004.

(23) Rebaudengo, M.; Sonza Reorda, M.; Torchiano, M.; Violante, M. Soft-error detection through software fault-tolerance techniques. In *DFT '99, Proceedings of the 14th International Symposium on Defect and Fault-Tolerance in VLSI Systems, Albuquerque, NM, Nov 1−3, 1999*; IEEE Computer Society: Washington, DC, 2002; pp 210−218.

(24) Carlson, W. W.; Draper, J. M.; Culler, D. E.; Yelick, K.; Brooks, E.; Warren, K. Introduction to UPC and language specification, 1999. The George Washington University High Performance Computing Laboratory. http://www.gwu.edu/upc/publications/upctr.pdf (accessed July 28, 2010).

(25) Numrich, R. W.; Reid, J. *SIGPLAN Fortran Forum* **2005**, *24*, 4–17.

(26) Čìžek, J. *J. Chem. Phys.* **1966**, *45*, 4256–4266.

(27) Paldus, J.; Čìžek, J.; Shavitt, I. *Phys. Rev. A* **1972**, *5*, 50–67.

(28) Paldus, J.; Li, X. Z. Critical assessment of coupled cluster method in quantum chemistry. In *Advances in Chemical Physics*; Prigogine, I., Rice, S. A., Eds.; John Wiley & Sons, Inc.: New York, 1999; Vol 110, pp 1−175.

(29) Bartlett, R. J.; Musiał, M. *Rev. Mod. Phys.* **2007**, *79*, 291–352.

(30) Kobayashi, R.; Rendell, A. P. *Chem. Phys. Lett.* **1997**, *265*, 1–11.

(31) Nieplocha, J.; Harrison, R. J.; Littlefield, R. J. *J. Supercomput.* **1996**, *10*, 169–189.

(32) Kendall, R. A.; Aprà, E.; Bernholdt, D. E.; Bylaska, E. J.; Dupuis, M.; Fann, G. I.; Harrison, R. J.; Ju, J.; Nichols, J. A.; Nieplocha, J.; Straatsma, T. P.; Windus, T. L.; Wong, A. T. *Comput. Phys. Commun.* **2000**, *128*, 260–283.

(33) Weikum, G.; Vossen, G. *Transactional information systems: Theory, algorithms, and the practice of concurrency control and recovery*; Morgan Kaufmann: San Francisco, CA, 2002.

(34) Dunning, T. H., Jr. *J. Chem. Phys.* **1989**, *90*, 1007–1023.

(35) Chinook SuperComputer. http://www.emsl.pnl.gov/capabilities/computing/ (accessed Apr 26, 2010).

(36) Voltaire Technologies. http://www.voltaire.com/ (accessed Sep 27, 2010).

(37) Mellanox Technologies. http://www.mellanox.com/ (accessed Sep 27, 2010).

CT100439U