Scalable Fault Tolerance Using Over-Decomposition and PGAS Models

Abhinav Vishnu [#1] Hubertus van Dam [#2] Wibe de Jong [#3]

[#1] *Computer Science and Mathematics Division, Pacific Northwest National Laboratory*
[#2,3] *Computational Chemistry Group, Pacific Northwest National Laboratory*
[1] abhinav.vishnu@pnnl.gov
[2] hubertus.vandam@pnnl.gov
[3] abhinav.vishnu@pnnl.gov

***Abstract*—Component failures in high-end systems are increasingly a norm rather than an exception. While application-transparent and application-aware approaches for check-point/restart have been proposed in the literature, they cease to scale beyond a few hundred nodes. Although, SSDs/NVRAMs alleviate the check-pointing overhead, the cost of recovery from failures is still proportional to the system size and not to the degree of failure. These properties are clearly prohibitive for the scale of the systems on the horizon.**

This paper is a divergence from the classical fault tolerance methods whereby the proposed approach uses a combination of Partitioned Global Address Space (PGAS) models to abstract the replication of critical data structures, and over-decomposition of computation in to smaller *tasks* for fine grained recovery. The proposed Scalable Fault Tolerance (SFT) approach uses the the *critical data* classified in read-only/read-write categories, automatically performs selective replication on these data structures, and uses a fault-tolerant meta-data store for recovery during faults. The SFT approach guarantees that cost of recovery is truly proportional to the degree of failure at the expense of very low time-space complexity. This is achieved by re-executing only the number of tasks upper bounded by the number of process failures and performing "continued execution" without re-spawning new processes. Using NWChem - a highly popular computational chemistry application - as a case study, the efficacy of the proposed solution is demonstrated on a commodity cluster with InfiniBand and actual process faults.

I. Introduction

Component failures in today's high-end systems are a major impediment to high supercomputer utilization. With much larger scale systems on the horizon, the mean time between failures (MTBF) of these systems is expected to decrease very significantly.

Legacy methods for fault tolerance include application-transparent approaches, under which the complete application state is saved to a fail-safe permanent storage. Kernel modules such as Berkeley Lab Checkpoint/Restart (BLCR) proposed by Hargrove *et al.* allow an application to transparently save the image to storage [1]. However, as shown by Gao *et al.*, the size of the checkpoint is in the order of Gigabytes/process using these kernel modules, even for a moderate system size [2]. The recovery method typically requires the job to be restarted from a previously saved checkpoint, resulting in the recovery cost being proportional to system size. Pro-active fault tolerance approaches such as Xen on RDMA enabled interconnects proposed by Liu *et al.* [3] promise to reduce the recovery cost, although at the accuracy of fault prediction. The pro-active approaches are also prohibitive at scale due to increased data movement of *virtual machine images* and associated energy costs. Message logging based approaches proposed by Cappello *et al.* reduce the cost of checkpointing for hybrid systems [4], [5], however, the cost of recovery is still proportional to the system size and not to the degree of failure. Additionally, message logging is useful in applications with temporal computation and communication patterns, which is not a characteristic of applications considered in this work.

As a result, Algorithm Based Fault Tolerance (ABFT) based approaches are becoming increasingly popular, although they require more involved design of check-pointing and recovery procedures. ABFT reduces the overall *checkpoint* space by identifying the *critical data structures*, and saving a copy in the memory hierarchy. The fault recovery is attempted by restarting the application from previously saved checkpoint. The checkpoint size is typically much smaller, and hence the overhead is relatively orders of magnitude lesser than the application-transparent approaches. However, the cost of recovery is still proportional to the *system size*, since the job still needs to be re-started from a previously saved checkpoint. Much of the previous research has primarily focused on efficient check-pointing mechanisms, such as recently proposed Scalable Checkpoint Restart Library by Moody *et al.* [6].

However, recent literature has shown that with increasing scale, the cost of recovery dominates the cost of check-pointing [7]. Hence, a methodology for fault tolerance at extreme scale must reduce the cost of recovery from failures. The recovery cost can be drastically reduced by decreasing the amount of re-computation at the point of failure. This can be achieved by *over-decomposition* of data and computation in *tasks* - sequential units of computation with input, output and optional dependencies. Task based execution models are finding increasing adoption in linear algebra libraries [8], [9]. These models are a divergence from legacy BSP models, as the data and associated computation are de-coupled from each other. A task may be scheduled on any process, and computed as soon as its dependencies are satisfied. The data requirements (input/output) to task models are typically described using Partitioned Global Address Space (PGAS) models [10], [1],

[11], [12], which provide one-sided mechanisms for accessing distributed data structures.

This paper proposes a novel approach for fault tolerance, where the over-arching objective is to have cost of recovery in presence of failures to be truly proportional to the degree of failure, rather than the system size. To this end, the proposed Scalable Fault Tolerance (SFT) approach extends Partitioned Global Address Space (PGAS) [10], [1], [11], [12] models for automatic replication of data based on its properties (read-only, read-write) and completely abstracts accesses to this fault tolerant data store. Such an abstraction is useful in designing selective replication algorithms (such as reed-solomon encoding for read-only structures) [13], [9] based on a combination of data properties and architecture (such as presence of SSDs/NVRAM). The automatic replication is combined with task based execution models, which leverage the fault tolerant data store for accessing read-only/read-write structures. At the point of failure, only the task(s) which were currently being executed by processes on the *faulty node* need to be re-executed - making the cost of recovery **truly proportional to the degree of failure**. A combination of PGAS and task models facilitates divergence from the fixed process set model - no additional processes need to be re-spawned at the point of failure. These properties makes the proposed SFT approach a very attractive solution for addressing hard faults at scale.

A. Contributions

Specifically, the contributions of the paper are:

- Proposition of a Scalable Fault Tolerance (SFT) approach, which provides automatic data replication atop PGAS models using data properties (read-only, read-write). The proposed approach uses Global Arrays for demonstration, however, it is readily extensible to other PGAS models.
- Fine-grained recovery requiring only re-execution of tasks which are bounded by the number of process failures. The SFT approach performs "continued execution" with the existing set of processes, facilitated by a combination of PGAS and task models.
- Detailed time-space complexity analysis of the SFT approach in presence of different replication strategies, and probabilistic analysis of multi-node failures to address the limitations of the SFT approach. Further reduction of recovery cost by accelerating the fault information propagation using flexible and scalable dissemination approaches.
- Integration of the proposed infrastructure with NWChem [14] - a highly scalable and popular computational chemistry package. An implementation of the proposed infrastructure and its evaluation with NWChem using actual process failures shows the efficacy of the proposed approach.

For many input decks with NWChem using up to 4096 processes and actual process failures, $< 15\%$ overhead is incurred in terms of execution time. This time includes the cost for writing to replicas, fault detection of processes and *continued execution with partial re-execution of tasks* in presence of failures. While the SFT approach has been demonstrated with one production application, it has potential to improve the resiliency of other applications and intermediate solver libraries [9].

The rest of the paper is organized as follows: section II presents the related work. Section III presents the background of the proposed approach. Section IV presents the solution space, followed by performance evaluation in section V. Section VI presents the conclusions and future directions.

II. Related Work

Multiple researchers have studied the approaches for designing fault tolerant algorithms, programming models and communication runtime systems [15], [16], [14], [2], [13], [17], [18], [19], [20], [21]. Approaches for network fault tolerance have been considered by many researchers as well. [18], [22].

Efficient scheduling algorithms for task based execution has been proposed by many researchers [14], [23], [24], [25]. Some scientific domains have a natural fit for task based execution, such as computational chemistry, and bio-informatics. There has also been a lot of research on formulating the problems primarily considered SPMD ambivalent to task based execution [9]. Dongarra *et al.* have proposed a broad variety of linear algebra kernels as task based scheduling problems and presented the performance advantages of this formulation. For most of the linear algebra kernels, the problem of scheduling is simplified due to a regularity in task dependencies [9], [8].

Sadayappan *et al.* have proposed framework for task based scheduling and fault tolerance [26], [27], [28]. However, these approaches assume a fixed process model. In these approaches, when a new spare node is introduced on failure, each and every task with an output to the spare node needs to be re-executed. In our proposed approach, only the task(s) which were being executed on the faulty node need to be re-executed. The number of tasks executed on a node at the point of failure is bounded by the number of processes, hence the recovery time of the proposed approach is *not* truly proportional to the degree of failure.

Fault tolerance with Message Passing Interface (MPI) has been tackled by multiple researchers [29], [30], [16], [31], [17], [3], [2]. Dongarra *et al.* have proposed multiple process models on failure using FT-MPI framework [13]. These approaches include re-spawning of MPI processes up on failure and patching them with the original communicator or continuing with holes (created by dead processes) in the communicator. The proposed SFT framework in this paper uses the second approach - to continue with the existing number of nodes. This approach is incorporated in the Global Arrays [10] communication runtime system, Aggregate Remote Memory Copy Interface (ARMCI) [32].

Gropp *et al.* discussed that the statement "MPI is not fault tolerant" is unfounded, since fault tolerance is a property of the combination of an MPI program and MPI implementation [16]. While most MPI implementations choose to abort on a fault

(assuming a checkpoint/restart methodology in place), this behavior is not mandated by the MPI standard. Gropp *et al.* also discussed methods of recovery using dynamic process creation and inter-communicators [16].

Fault tolerance using application transparent and assisted approaches have been discussed widely by multiple researchers [2], [33], [31], [17] . Gao *et al.* have presented that user-transparent check-pointing using Berkeley Lab Checkpoint Restart (BLCR) is beneficial for NAS Parallel Benchmarks [34] and checkpoint aggregation can reduce the overall time of check-pointing significantly. However, the overhead increases significantly with increasing number of processes [2].

Cappello *et al.* have shown that pessimistic message based logging may be used for recovery on volatile nodes using MPICH-V [31], [17]. However, this approach is not applicable to PGAS models like Global Arrays [10], as it assumes a two-sided execution model. Message Logging reduces the check-pointing overhead, but still results in the cost of recovery to be proportional to the system size. Another benefit of the SFT framework proposed in this paper is its continued execution.

Many researchers have also presented fault tolerance methods using virtual machines. Recent work includes Xen over high speed networks [3], in addition to the classical work based on Virtual Machines using TCP/IP. However, these approaches either require check-pointing for post failure execution [35] or require pro-active fault tolerance [3]. With pro-active fault tolerance, the success depends on the accuracy of fault prediction. The proposed approach in this paper provides continued fault tolerant execution, without any dependency on the accuracy of fault prediction.

Computational chemistry codes typically checkpoint intermediate results to stable storage for restart on node failure(s). This approach is suitable for iterative calculations including self consistent field, and Coupled Cluster (CCSD) methods. However, computationally expensive parts of calculation - Triples energy correction of Coupled Cluster with perturbative triples (CCSD(T)) are non-iterative in nature. As a result, there are no natural points of check-pointing, while the probability of failure during the execution of this method is high, due to its high computational complexity.

III. Background

This section presents background of our work. A brief description of over-decomposition in the context of this paper, followed by a description of the Partitioned Global Address Space (PGAS) models.

A. Over-decomposition and Task Based Execution Paradigm

Over-decomposition is a strategy to further decompose the data (and potentially computation on the data) available to a process in multiple smaller chunks. This methodology is beneficial for better cache utilization, and load balancing. It is important for the chunk to be large enough, so that the cost of load-balancing (potentially off-node data movement) is offset by the work balancing. In this context, task-based execution models have gained momentum, which use data and computation over-decomposition for load balancing and resilience.

A task is a unit of computation with an input (potentially from multiple processes), output (potentially to multiple processes) and dependencies. Each task can be described using meta-data information for input, output and dependencies. A task may be scheduled on any process, as soon as its dependencies are satisfied. Independent tasks are a special case, where a task may be scheduled as soon as a process/thread can execute it. MapReduce paradigm is an example of independent tasks, with "NULL" output to global address space data. The proposed SFT framework focuses on tasks, which have output to large number of processes scheduled on multiple distributed memory nodes. Figure 1 shows examples of independent tasks, tasks with dependencies and tasks which may produce more tasks. The proposed SFT approach is applicable to each of these task models.

B. Partitioned Global Address Space Models

Partitioned Global Address Space programming models provide an abstraction of globally distributed memory space. In addition, the PGAS models provide interfaces for accessing local data directly for minimizing data movement.

Many language and library implementations are being designed and implemented under this model such as X10 [11], Chapel [12], High Performance Fortran, Unified Parallel C (UPC) [1], ZPL, and Titanium. These languages provide load/store mechanisms for global address space data. They also provide efficient runtime implementations to coalesce multiple such data accesses by resolving write dependencies. PGAS models provide data centric abstractions, and decouple the static scheduling of computation from data. This property of PGAS models is used to provide fault resiliency in the proposed SFT approach. Library based implementations such as Global Arrays [10] and SHMEM [36] have also been designed and implemented to serve the purpose of remote memory accesses.

Figure 2 shows an example of virtual and physical distribution of a global array. The figure is used to show the virtual representation of 16x16 array on 4 processes.

IV. Fault Tolerance Solution Space

This section presents elements of the solution space for the proposed SFT approach. The section begins with a description of the preliminaries, time-space complexity analysis, discussion on handling multiple node failures and concludes with a putting it all together with NWChem [14] section.

A. Preliminaries

Any fault tolerance solution typically has many catastrophic scenarios. Hence, it is important to underline the assumptions as precisely possible. In this paper, it is assumed that the node/process failure is *permanent*. The point of fault discovery may be far apart for the processes in a job, but once a process is *discovered as dead*, it is considered dead for rest of the program execution.

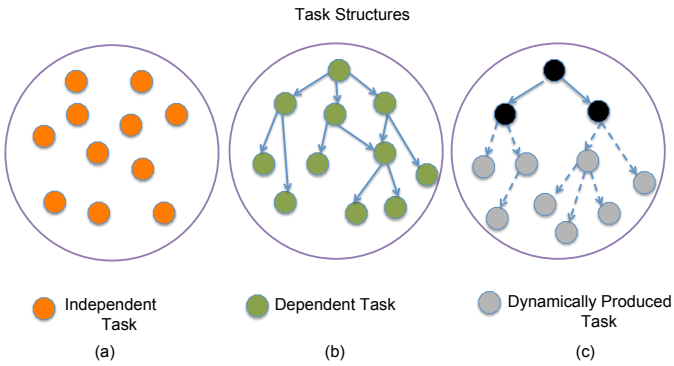

Fig. 1. Task Structures: Examples with independent tasks, tasks with dependencies, and tasks which may produce more tasks

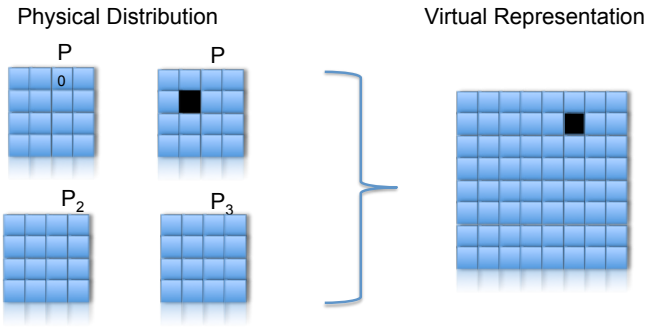

Fig. 2. Examples of physical and virtual distribution of a global arrays in PGAS models

It is also assumed that a process failure does not abort the job. A job abort on failure is not a problem for checkpoint/restart methods, as they rely on restarting the computation from a previously saved checkpoint. The SFT approach performs "continued execution" in the presence of node faults - the application continues execution without a need to re-spawn new processes. This property is a critical element of the proposed design, since this allows the cost of recovery to be low.

High-end systems such as IBM Blue Gene based systems shut down the midplane on process/node failures. Similarly, Cray XT/XE/XK systems abort the job as soon as the ALPS scheduler discovers a process/node failure. With these limitations, the proposed approach is designed and evaluated on commodity systems based on InfiniBand [37]. However, the proposed approach can be readily executed on high-end systems, as soon as the restrictions on from the process manager are lifted.

B. Critical Data for Replication

Critical Data is the data structure set, which must be saved/replicated, so that it may be used during fault(s). The critical data structure is a property of the programming model and the fault tolerance algorithm.

In application-transparent methods of checkpoint/restart, complete process space must be saved, as there is no awareness on the criticality of any data structure. This methodology ceases to scale beyond a few hundred nodes, due to the pressure on I/O subsystem. The advent of disruptive technologies such as non-volatile RAM/SSDs may alleviate the cost of application-transparent check-pointing. However, the all the processes need to be restarted from a previously saved checkpoint.

An alternative and popular approach is application-aware methods for checkpoint/restart. This approach relies on critical data identification by (potentially) involvement of a domain expert. The critical data is saved at intermediate points (typically at control synchronization), and the restart algorithm requires re-spawning of processes to read from the saved critical data. With this approach, the overall volume of the critical data is typically reduced by orders of magnitude in comparison to the approach presented above. However, the cost of recovery is still proportional to the system size, as new processes need to re-spawned.

The overall volume of critical data in application-aware checkpoint methods is a property of the algorithm. Using *dgemm* ($C = A \cdot B$) as an example, there are multiple choices of replication of read-only matrices A and B matrices. Many proposed algorithms in literature use reed-soloman encoding for read-only matrices. These approaches reduce the overall volume of critical read-only data $\in \Theta(N)$, while full replication requires $\Theta(N^2)$, where N is the dimension of the matrix. However, full replication is needed for C matrix, as it is updated asynchronously by different processes.

Since many algorithms operate on dense/sparse matrices, it is worthwhile to unify their distribution across processes by using Partitioned Global Address Space (PGAS) models. These models abstract the data distribution such that a process may operate on distributed data using get/put/accumulate, and the local data using load/store semantics.

In the proposed SFT approach, an application only needs to specify whether a particular PGAS data structure is critical and its attributes (read-only/read-write). The PGAS runtime automatically uses no-replication (if the structure is not critical), reed-soloman encoding (if the structure is critical and read-only), and full replication (if the structure is critical and write-only).

Another advantage of using PGAS models is automatic distribution of checksums based on reed-soloman encoding, and automatic re-distribution of these checksums, when faults occur. The PGAS models are also capable of abstracting deep levels of memory hierarchy (such as byte addressable SSDs), and intelligently storing replicas for reuse during failures.

C. Over-decomposition and Task Models

The replication strategy presented above is useful in the *dgemm* example, where each of the matrices may be appropriately marked as critical, read-only/read-write. Since C is completely replicated, it is essential to guarantee that at least one of its replicas are in consistent state at the point of recov-

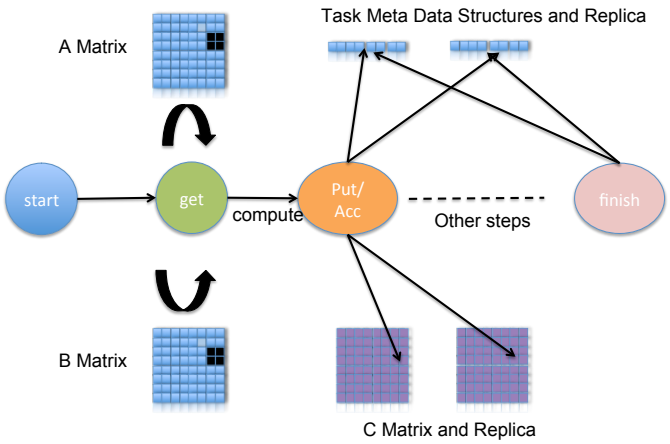

Fig. 3. Deterministic Finite Automata for various task states: A task starts and read(s) patches of A and B matrices, updates the task meta-data structures, performs computation, and updates the C matrix. Additional states may be defined up on the application (denoted by dotted lines). The finish state is also updated using transactional semantics.

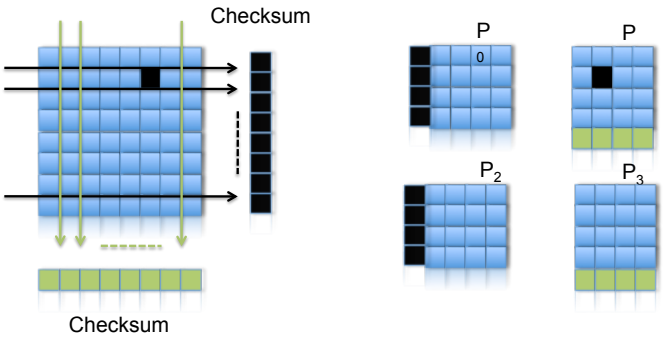

Fig. 4. Checksums using reed-soloman encoding and their distribution across processes

ery. Hence, it is importance to use the transactional memory update semantics for read-write critical data structures.

With transactional semantics, each of the replicas must be updated sequentially. Most PGAS models support location consistency, which is insufficient for transactional update semantics [38]. In the proposed approach, each update is fenced to the remote process - each update guarantees that the data is written to the remote memory.

Another important element of the proposed approach is the need for a fault tolerant meta-data for critical data structures. The meta-data is important to mark the state of a data critical structure - whether it is *dirty* or *consistent*. The state of a critical data structure may be dirty, if another process could not complete update to this data structure before failing. The associated meta-data structure may be used to update the state, to see whether the critical data structure is dirty or consistent. This meta-data structure is a critical read-write structure in itself. It uses the PGAS infrastructure presented above for complete replication.

A simplified and yet powerful use case is *independent tasks*, where tasks may be executed as soon as they are generated. Task based execution models have been used in Linear Algebra (DaGue) [8], and full applications such as NWChem [14], which rely on get-compute-put task model for load balancing reasons. An advantage of using over-decomposition for fault tolerance is a potential for finer grained recovery.

Underlying runtime(s) of task based execution models are geared primarily for performance. Using the classical *get-compute-put* model with task models, it is important that each of the states in *get-compute-put* model are fault tolerant. Figure 3 shows an example of combining automatic replication of PGAS structures with additional task states for fault tolerance. The figure uses $dgemm$ as an example.

A task starts with the *get* state, reading patches from A and B. No additional states are needed for *get* state, because the underlying PGAS infrastructure automatically provides the patch. Updates to the task meta-data structure are not required during the *get* phase, because the state of any critical data structure is not updated. Similarly, the compute phase does not require an update of the task meta-data structure. Each update to a replica requires that the task meta-data is updated to guarantee the consistency of the replica. Once the updates to each replica are completed, the task meta-data structure is updated to *finish* state.

The proposed approach requires additional state management and replication and transactional updates to read-write data structures. Hence, it is important to understand the time-space complexity of the proposed approach. This is presented in the upcoming section.

D. Time-Space Complexity Analysis

Table I shows the attributes, which are used to model the space and time complexity of the proposed approach.

	Property	Symbol
1	Message Size for Data Transfer	m
2	Total Number of Processes	p
3	Number of Processes/Node	c
4	Number of replicas	k
5	Number of critical read-only data structures	α
6	Number of critical read-write data structures	β
7	Number of total tasks	δ

TABLE I
TIME AND SPACE ATTRIBUTES FOR FAULT TOLERANCE APPROACH

Since most of the computation is performed on matrices, the space complexity is modeled using the matrix dimension (N). The total number of elements in a matrix is N^d for a d dimension matrix, assuming a dense data representation, although the analysis is readily extensible to sparse matrix representations.

Let M_{ro} represent the space complexity of the critical read-only data structures. Using reed-soloman encoding for read-only structures (row and column encoding), the space complexity for read-only structures is:

$$M_{ro} = k \cdot \alpha \cdot (d-1) \cdot N/p \quad (1)$$

The above equation is a model, when the checksum generated using a reed-soloman encoding is distributed equally among processes. The space complexity is proportional to the number of dimensions, the number of critical read-only structures and the size of the dimension itself. For a d dimension matrix, the size of the checksum is proportional to $(d-1)$. With reed-soloman encoding, M_{ro} scales very well in comparison to the original space complexity ($\alpha \cdot N^d/p$).

Let M_{rw} represent the space complexity of critical read-write data structures. These data structures need to be replicated completely, as they would be updated asynchronously.

$$M_{rw} = k \cdot \beta \cdot N^{d-1}/p \quad (2)$$

From the above equation, it is clear that the proposed approach is feasible when the size of the critical read-write data structures is smaller than the total memory available to a process. There are scenarios, where this is possible, such as strong scaling of an application where the time to solution is a primary evaluation criteria. Weak memory scaling experiments may not be able to utilize the SFT approach proposed in this paper.

Another important aspect is the value of k. The number of replicas can be decided by the amount of memory available per process, and the volume of data that has to be updated. A single replica increases the MTBF of the application from $1/p$ to $1/\sqrt{p}$, which could be sufficient increment for an application to complete its execution.

Let M_t represent the space complexity of meta-data structure which holds the state of the tasks. M_t is given by:

$$M_t = k \cdot \delta/p \quad (3)$$

δ is defined by the granularity of the task decomposition of the application. However, even for moderate sizes of d, $M_t \ll M_{rw}$. Hence, the space complexity of the proposed approach is dominated by M_{rw}. Next, the time complexity of the proposed approach is presented.

Let T_{ro} represent the time complexity of reading the read-only data structures. In the proposed approach, the read-only data structure needs to be read from exactly one replica. As a result, the T_{ro} is similar in comparison to the original time complexity.

Let T_{rw} represent the time complexity of updating the read-write data structures. There are two read-write structures - the application data and the task meta-data. The task meta-data is small - updating the task meta-data is latency bound. The time to update the application data is dependent on the volume of the data, and the size of each update (to determine whether it is latency/bandwidth bound). The worst case scenario occurs when each update is latency bound. For purpose of simplification, the time to update the meta-data is ignored, since the overall volume of read-write structure to be updated for each task is much larger.

Using $dgemm$ as an example, it is possible to determine the time-complexity of the proposed approach. Each task reads from two critical read-only data structures (A and B) , and updates one critical data structure (C). For δ number of tasks, the average work performed by each task is N^3/δ. Each task needs to update $k \cdot N^2/\delta$ volume of data to all the replicas. For an entirely latency bound transfer with l average latency, the time taken for the above step is $k \cdot l \cdot N^2/\delta$. For simplicity, with one replica, the cost reduces to $l \cdot N^2/\delta$. While l is in μs, in the best case scenario each computation can be performed in a single clock cycle.

Let o_{cr} represent the check-pointing overhead to replicas. $o_{cr} = (l.N^2/\delta)/(N^3 * cpu_{freq}/\delta) = l/(N * cpu_{freq})$. Using conservative values of l as 10μs and cpu_{freq} as 2 GHz, $o_{cr} = 10.10^3/(N*2) \approx 10^5/N$. For overhead to be less than 0.1, N should be at least 10^6 elements, which is a relatively small problem size for $dgemm$ at scale. Since the overhead is inversely proportional to N, the increasing problem size reduces the check-pointing overhead.

Let o_{re} represent the recovery overhead per node failure. Under the proposed approach, only those tasks need to be re-executed, which were currently being executed by the processes before the node failure. Hence, $o_{re} = c \cdot N^3/\delta$. o_{re} is inversely proportional to the number of tasks - finer grained tasking would be conducive for the proposed approach.

The time-space complexity analysis gives a good understanding of the applicability of the proposed approach. Similar analysis may be used by applications to select values of different parameters. The next section is dedicated to addressing a limitation of the proposed approach.

E. Handling Multiple Node Failures

Theorem 1. *For n nodes and k replicas, the probability of catastrophic failure due to unavailability of all replicas is $\in O(1/(n-k)^k)$ and $\Omega(1/n^k)$.*

Proof: The probability of a node failure is $1/n$. The conditional probability of a replica failure is $1/n \cdot 1/(n-1)$. Extending the conditional probability to k replicas, the probability of failure of each of the replicas is:$1/n \cdot 1/(n-1) \cdots 1/(n-k)$ which is $\in \Omega(1/n^k)$ and $O(1/(n-k)^k)$. ■

It is easy to see that the probability of a catastrophic failure with k replicas is very low. Even a single replica increases the application MTBF from $1/n$ to $1/\sqrt{n}$ - for k replicas, the application MTBF increases to $1/n^{1/k}$.

Using this analysis, the benefits of designing algorithms to handle multiple node failures may not be beneficial due to reduced probability. Additionally, the proposed approach can still handle many node failures, as long as all the replicas have not been destroyed.

F. Detailed Design Issues

The proposed framework for fault tolerance is a clear divergence from classical methods - re-spawning of new processes is not required. Due to semantic mis-match between the requirements of the framework and MPI [29], [30] specification, a new *fault tolerance management infrastructure (FTMI)* is proposed in the previous work by Vishnu *et al.* [15]. The major elements of FTMI are scalable and hardware assisted fault detection, and fault tolerant collective communication for control synchronization. PGAS applications use load/store semantics for reading/writing data to various data structures, and only control synchronization is needed, in addition to fault tolerant data fence.

Hereby, an additional element of the proposed SFT infrastructure is faster fault information propagation. With increasing scale, it is important to accelerate fault detection, so that each process does not need to wait for a *timeout* to detect a process/node failure. A conservative value of timeout has to be used with increasing scale to prevent false positive - making the problem even harder with scale.

To facilitate this, a data structure to represent the alive/dead status of other nodes is used and shared among processes on a node. The base address of this data structure is exchanged at the startup, so that any process may write to this area of memory directly on fault detection. This is equivalent of a broadcast, which has a time complexity of $\log(p)$ using the k-nomial algorithms [29], [30].

G. Putting It All Together: A Case Study with Chemistry

This section uses the proposed approach above to design a fault tolerant triples correction module within a high performance and scalable computational chemistry code - NWChem [14]. NWChem uses PGAS data structures based on Global Arrays [10] and task based models for load balancing among processes. The proposed approach presented in the previous sections is implemented with Global Arrays, as much of the original algorithm uses Global Arrays and task based models.

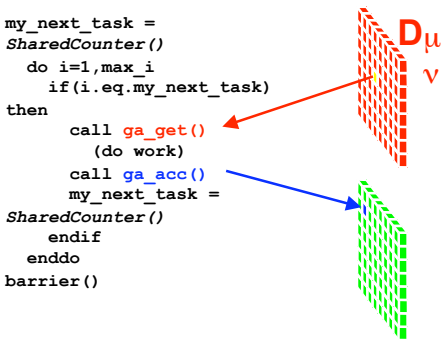

Fig. 5. Chemistry Calculation Schematic Diagram

Figure 5 is an abstraction of many computational chemistry algorithms such as self-consistent field, singles, double and triples energy calculation. These algorithms are ordered in terms of accuracy and computational complexity. The key elements of the computation are a *shared counter*, which is a load balance counter representing a task id of *my next task*. The task id is a variable in the indices of global arrays to determine the patch of a global array to be read/updated. Since each task id determines which indices to read, get/put model is useful for these calculations. The figure shows a case, when the output of *do work* has to be *accumulated* in a global array. The *do work* involves multiple calculations including *dgemm* and input/output to multiple processes.

At this point, it is worthwhile noting that the problem is a superset of MapReduce paradigm, where each mapper never updates any global memory directly. As shown in this diagram, it is the asynchrony of the calculation in get/accumulate, which makes the problem harder and the MapReduce paradigm is insufficient to solve this problem.

The demonstration case in this paper is triples energy calculation. The triples calculation is challenging for two reasons - it is the most computationally phase of the application, using as much as 90% of the compute time, and it is inherently difficult to make this fault tolerant, because it is non-iterative in nature [39]. **No known fault tolerant algorithm, even based on checkpoint/restart exists for triples**.

There are several elements of the algorithm. The SharedCounter is a load balance counter, which may be hierarchically distributed, and it needs replication. However, for legacy reasons, the counter is resident on the first node. This limitation can be addressed by requiring a process to compute a task id equivalent to its MPI rank in the beginning, and then requesting the shared counter at a later point. The other elements are fault tolerant get and accumulate. Each of these operations are un-changed, since the replication of these critical data structures is abstracted from the application.

Another important element is *termination detection* - who should re-execute incomplete tasks. Since the task meta data is fully replicated, at least one copy exists which correctly shows the state of each task. For k replicas, a simple protocol is used, in which *responsible* processes are given an execution priority. The process, which has the highest priority re-executes the incomplete tasks. This approach is used to design the fault tolerant triples algorithm (FT-Triples). The other elements for fault tolerant execution, such as FTMI are integrated with the Global Arrays runtime.

V. Performance Evaluation

This section presents a performance evaluation of the FT-Triples algorithm using an InfiniBand cluster. Multiple input decks consisting of smaller calculations and larger calculations of two water molecules and uracil molecule are used to evaluate the performance of the algorithm. For each of these input systems, the performance of the Original implementation - the official 5.0 release of Global Arrays [10] is compared with FT-Triples implementation for no-faults and multiple node faults during the execution.

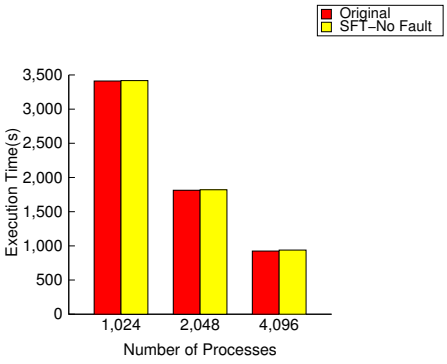

Fig. 6. Uracil, 1 Molecule

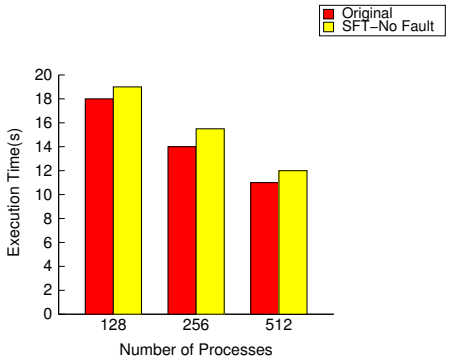

Fig. 7. Water, 1 Molecule

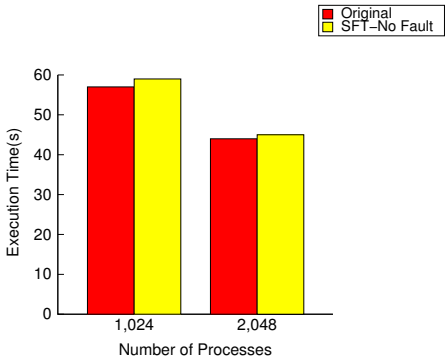

Fig. 8. Water, 2 Molecules

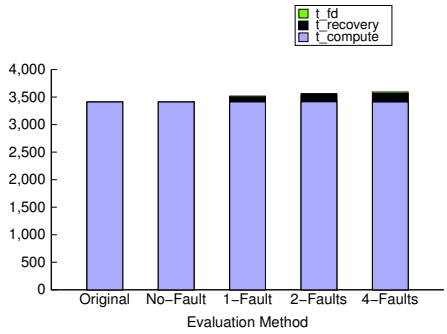

Fig. 9. Uracil, 1024 Processes

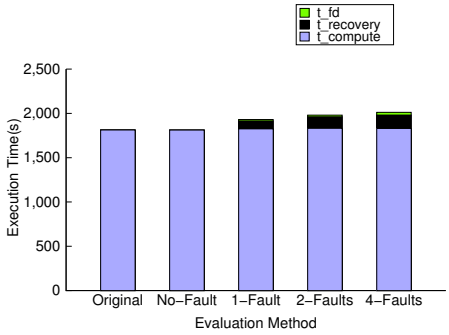

Fig. 10. Uracil, 2048 Processes

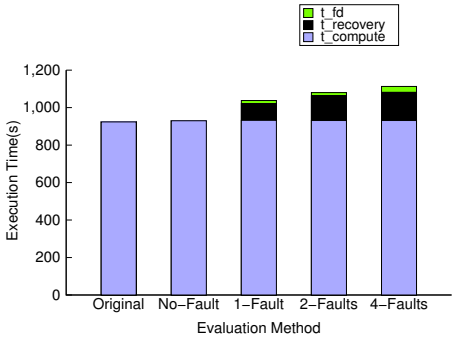

Fig. 11. Uracil, 4096 Processes

A. Experimental Testbed

The FT-Triples algorithm is evaluated on Chinook - a supercomputer at Pacific Northwest National Laboratory. Chinook [40] is a 160 TFlops system that consists of 2310 HP DL185 nodes with dual socket, 64-bit, Quad-core AMD 2.2 GHz Barcelona Processors. Each node has 32 Gbytes of memory and 365 Gbytes of local disk space. Each node is interconnected with InfiniBand DDR network using Mellanox InfiniBand Network Interface cards and Voltaire switches.

B. Evaluation Methodology

An important aspect of our performance evaluation the characterization of performance with actual process faults. The fault is injected using a helper thread and executing `system(``kill -9 -1'')` on the node. While the node is still alive in true sense, the processes are dead, and as a result the associated computation and memory is destroyed. On production systems with only user-level accesses, this is as close as possible to stress test the performance of the proposed approach for actual node failures. The number of failures is varied from 1-4 to capture multi-node failure scenarios.

C. Empirical Space Complexity

The space complexity utilization of critical data structures in NWChem can be defined using number of basis functions (b), number of orbitals (o), number of molecules (m) and number of un-occupied orbitals (u). Typically, $u = b - o$. Let M_{nd} define the space complexity of NWChem on a particular input deck. M_{nd} = $m^4 \cdot u^2 \cdot o^2$ doubles [39].

The performance evaluation uses Uracil molecule, which has 340 basis functions, and 29 occupied orbitals. The overall space complexity is 620 MB. This input deck can be used for strong scaling up to 4096 processes. Using a conservative estimate of 1 GB/process, the size of the critical data structure is $< 1\%$ of the overall available memory. The rest of the memory is for calculating local integral transformations. These calculations do not affect any non-idempotent data structures and can be re-calculated as a part of the compute phase of a task.

The performance evaluation also uses multiple Water molecules for performance evaluation, which has 51 basis functions, 5 occupied orbitals. The space complexity is $\approx m^4 \cdot 2$ Mbytes. The 24 water molecule used at full Jaguar scale [39] uses $\approx$ 50 Gbytes of total memory - $< 1\%$ of the total available memory. Here, we use two water molecules are used for evaluation, because of the much smaller size of the evaluation testbed.

D. Empirical Time Complexity

The empirical performance evaluation is performed with Uracil and two water molecules, as discussed above. The performance evaluation is done using one replica.

Each empirical time evaluation is further divided in multiple elements:

- t_fd: Time for fault detection
- $t\_compute$: Time for computation, excluding recovery

- $t\_recovery$: Time spent during recovery phase, including re-computation of tasks

1) Overhead Without Faults: Figures 6, 7 and 8 show the performance comparison of the original implementation with the SFT implementation using Uracil, Water (1 Molecule) and Water (2 Molecules), respectively. The overhead present in the no-faults case is a reflection of the extra communication which has to be done to the read-write replica.

The execution time for Uracil with Original implementation decreases linearly, due to the amount of work available to each process in number of tasks (δ). The overhead in no-faults case is $\approx 3\%$ in each of the cases - a very acceptable overhead. For significantly long running applications, the incurred overhead in absence of faults is acceptable. These results are 6x better than the results reported by Moody *et al.*, where only 85% of the system efficiency is available for long running applications due to check-pointing overhead [6]. The Water (2 Molecules) case

The Water (1 Molecule) and Water (2 Molecule) cases are short running input decks. The Water (1 Molecule) takes 18s on 128 processes, and adds about 15-20% overhead with the proposed SFT approach. For very short execution time cases, it is not worthwhile to use the SFT approach. It is expected that the system MTBF would be in order of tens of minutes for any realistic calculation to succeed on larger scale systems. Hence, this significant check-pointing overhead does not undermine the capability of the proposed SFT approach. The Water (2 Molecules) case gives an overhead of less than 5% on 1024 and 2048 processes, respectively.

2) Performance Evaluation with NWChem and Uracil: This section presents a performance evaluation of Uracil molecule with actual process faults. The 1-fault case is equivalent of one-node fault, which involves eight process failures. The evaluation uses up to 4-faults (equivalent of 32 process failure on 4-nodes). Figure 9, 10, and 11 show the result of the evaluation on 1024, 2048 and 4096 processes, respectively.

t_{fd} is relatively small in comparison to the overall execution time for this input deck. This is primarily due to a combination of accelerated fault detection using hardware assisted mechanisms and fault information propagation design. t_{fd} is expected to be $O(p)$, with high constant due to the large value of timeout. However, this is a very loose bound in practice, because fault information propagation using collectives can perform the broadcast in $log(p)$ [29], [30].

Table II shows the overhead in comparison to the original implementation. The 1024 process takes $\approx$ 3500s to execute with the Original implementation as shown in Figure 9. A failure of eight processes adds about 3% overhead in execution time, which is highly scalable in terms of execution time. As the number of failures increase, the overall overhead increases due to the total increment in fault discovery and re-execution. With 4-faults, the overhead is 6%, which is still very reasonable given the total execution time and the number of processes.

The 2048 process execution as shown in Figure 10 and table II shows the performance evaluation and overhead, respectively. The Original implementation takes $\approx$ 1850s to completion. The total overhead with No-Faults increases due to strong scaling, in comparison to 1024 processes. With 1-Fault, the overhead is 7%, which is 4% higher than 1024 processes. The increment in overhead is due to higher time spent in fault discovery and increased overhead of task re-execution. The total time to re-execute the tasks does not decrease proportionally, since there are only eight tasks to be re-executed (1-Fault is same as eight process failures resulting in up to eight task re-executions) with increasing number of processes, the overall execution time has decreased proportionally.

Similar increments in overhead are observed with 4096 processes using the table and the Figure 11. However, 4096 process case takes $\approx$ 900s. The overhead in presence of one fault is 11%. This is still 4% better than the state of the art SCR system, which provides up to 85% efficiency for applications with check-pointing [6]. The proposed SFT approach is still very scalable, as it needs to execute very few tasks, and the overhead of fault discovery is largely mitigated by accelerated fault information propagation.

Procs	No-Fault	1-Fault	2-Faults	4-Faults
1024	1	3	5	6
2048	2	7	9	11
4096	3	11	15	22

TABLE II
PERCENTAGE OVERHEAD COMPARED TO ORIGINAL IMPLEMENTATION WITH URACIL MOLECULE

3) Performance Evaluation with NWChem and Water Molecules: This section presents a performance evaluation of NWChem with Water molecules. Figures 12 and 13 show the performance of two Water molecules on 512 and 1024 processes, respectively. As the figures show, the overall execution time of the base case is $\approx$ 1 minute on 512 processes. The purpose of this test is to understand the feasibility of the proposed approach, when the overall execution time of the job is small.

Both charts show that the cost of fault detection is non-negligible for such small executions. The cost of fault detection is about 20% higher for 1024 processes in comparison to the 512 processes. For 512 processes, 1-Fault case increases the execution time by about 27%. As the number of faults increase, the overhead increases significantly, as the amount of computation is relatively small. Clearly, for such small executions, the SFT approach is not feasible. However, it is difficult to design any scalable methods for fault tolerance for such small executions. It is expected that the MTBF of the systems on horizon would be much larger than the execution time here. The strong scaling execution of two water molecules shows greater relative overhead as shown in Figure 13. However, the execution time of the base case is 45s, which is much smaller than the expected MTBF.

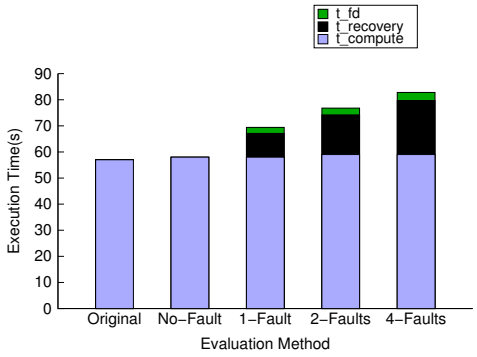

Fig. 12. Water, 2 Molecules, 512 Processes

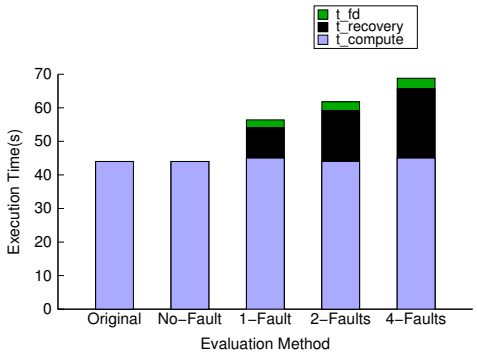

Fig. 13. Water, 2 Molecules, 1024 Processes

E. Possible Improvements and Other Considerations

There is scope for improving the cost of recovery by altering the check-pointing methods and adapting the algorithm. As an example, the percentage of critical memory is a small fraction of the overall memory available to a process. Hence, it is possible to completely replicate the read-only data structures and further reduce the cost of recovery. This would only nominally increase the space complexity of the SFT approach.

The primary limitation of the proposed approach is that the read-write data structures must be completely replicated. For some application domains such as ones which perform stencil computation, the complete memory space is a read-write data structure. In these circumstances, legacy methods of checkpoint/restart and message logging based approaches may be useful, since they have regular communication patterns. However, the proposed SFT approach is still applicable to a strong scaling evaluation of these application kernels.

Another possibility to reduce the recovery overhead is to reduce the granularity of each task. The NWChem application primarily uses the SharedCounter to determine the patch of global arrays to read/update. Depending on the task id, the overall time for executing a task may vary from a $10ms$ to 30s. The variance may be further reduced by algorithmic changes, and keeping a ratio of communication to computation time low, while allowing better load balancing and further recovery.

VI. Conclusions and Future Work

This paper is a convergence of fault tolerance approaches, where the cost of recovery in presence of failures is truly proportional to the degree of failure, rather than the system size. To this end, the proposed approach has used Partitioned Global Address Space (PGAS) models for automatic replication of data based on its properties (read-only, read-write) and abstracted much of the accesses to this fault tolerant data store. Such an abstraction is useful in designing selective replication algorithms (such as reed-solomon encoding for read-only structures) based on a combination of data properties and architecture (such as presence of SSDs/NVRAM).

The proposed approach has used over-decomposition of data and computation in *tasks* - units of computation, which may be scheduled on any process as long as the data dependencies are met. Task based execution models are finding increased adoption in linear algebra libraries, as they are conducive for dynamic load balancing. The inherent properties of tasks (get-compute-put) leverage the fault tolerant data store for accessing read-only/write-only structures. At the point of failure, only the tasks which were currently being executed by processes on the *faulty node* need to be re-executed - making the cost of recovery **truly proportional to the degree of failure**. A combination of PGAS and task models facilitates divergence from the fixed process set model - no additional processes need to be re-spawned at failure. These properties has made the proposed approach a very attractive solution for addressing hard faults at scale.

The proposed fault tolerance infrastructure is plugged in the most computationally challenging module of NWChem [14] - a highly scalable and popular computational chemistry package. An implementation of the proposed infrastructure and its evaluation with NWChem using actual process failures shows that $< 15\%$ overhead is incurred in terms of execution time. This time includes the cost for writing to replicas, fault detection of processes and *continued execution with partial re-execution of tasks* in presence of failures.

The on-going work is to study the sensitivity of NWChem and other applications to bit-flips in critical data structures. This study is an important step in defining fault tolerant algorithms, which can detect and correct slient errors using the automatic replication infrastructure presented here.

References

[1] P. Husbands, C. Iancu, and K. A. Yelick, "A Performance Analysis of the Berkeley UPC Compiler," in *International Conference on Supercomputing*, 2003, pp. 63–73.

[2] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand," in *International Conference on Parallel Processing*, 2006, pp. 471–478.

[3] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High performance vmm-bypass i/o in virtual machines," in *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*. USENIX Association, 2006, pp. 3–3.

[4] A. Bouteiller, B. Collin, T. Hérault, P. Lemarinier, and F. Cappello, "Impact of event logger on causal message logging protocols for fault tolerant mpi," in *IPDPS*, 2005.

[5] L. A. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: high performance fault tolerance interface for hybrid systems," in *SC*, 2011, p. 32.

[6] A. Moody, G. Bronevetsky, K. Mohror, and B. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *SuperComputing*, 2010.

[7] D. Fiala, F. Mueller, C. Engelmann, R. Reisen, K. Ferreira, and R.Brightwell, "Detection and Correction of Silent Data Corruption for Large Scale High Performance Computing," in *Supercomputing*, ser. SC'12, 2012.

[8] Jack Dongarra et al., "Directed Acyclic Graph Unified Environment," http://icl.cs.utk.edu/dague/.

[9] U. o. T. Innovative Computing Laboratory, "Numerical linear algebra on emerging architectures: The plasma and magma projects."

[10] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers," *Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2005, pp. 519–538.

[12] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *International Journal on High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[13] G. E. Fagg and J. Dongarra, "Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world," in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, London, UK, 2000, pp. 346–353.

[14] R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, and A. T. Wong, "High Performance Computational Chemistry: An Overview of NWChem, A Distributed Parallel Application," *Computer Physics Communications*, vol. 128, no. 1-2, pp. 260–283, June 2000.

[15] A. Vishnu, H. Van Dam, W. De Jong, P. Balaji, and S. Song, "Fault Tolerant Communication Runtime Support for Data Centric Programming Models," in *International Conference on High Performance Computing*, 2010.

[16] W. Gropp and E. Lusk, "Fault Tolerance in Message Passing Interface Programs," *International Journal on High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, 2004.

[17] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003, p. 25.

[18] A. Vishnu, A. Mamidala, S. Narravula, and D. K. Panda, "Automatic Path Migration over InfiniBand: Early Experiences," in *Proceedings of Third International Workshop on System Management Techniques, Processes, and Services, held in conjunction with IPDPS'07*, March 2007.

[19] A. Vishnu, B. Benton, and D. K. Panda, "High Performance MPI on IBM 12x InfiniBand Architecture," in *International Workshop on High-Level Parallel Programming Models and Supportive Environments, held in conjunction with IPDPS '07 (HIPS'07)*, 2007.

[20] A. Vishnu, P. Gupta, A. R. Mamidala, and D. K. Panda, "A Software Based Approach for Providing Network Fault Tolerance in Clusters with uDAPL Interface: MPI Level Design and Performance Evaluation," in *SuperComputing*, 2006, pp. 85–96.

[21] A. Vishnu and D. K. P. M. K. Krishnan, "A Hardware-Software Approach to Network Fault Tolerance wwith InfiniBand Cluster," in *International Conference on Cluster Computing*, 2009, pp. 479–486.

[22] A. Vishnu, M. ten Bruggencate, and R. Olson, "Evaluating the potential of cray gemini interconnect for pgas communication runtime systems," in *Proceedings of the 2011 IEEE 19th Annual Symposium on High Performance Interconnects*, 2011, pp. 70–77.

[23] 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, pp. 740–749, 2006.

[24] Subsurface Transport over Multiple Phases, "STOMP," http://stomp.pnl.gov/.

[25] L. G. S. Donfack and A. K. Gupta, "Adapting communication-avoiding LU and QR factorizations to multicore architectures," in *IPDPS*, 2010.

[26] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan, "Scioto: A framework for global-view task parallelism," in *ICPP*, 2008, pp. 586–593.

[27] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng, "Dynamic load balancing of unbalanced computations using message passing," in *IPDPS*, 2007, pp. 1–8.

[28] J. Dinan, A. Singri, P. Sadayappan, and S. Krishnamoorthy, "Selective recovery from failures in a task parallel programming model," in *CCGRID*, 2010, pp. 709–714.

[29] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[30] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. L. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: Extending the message-passing interface," in *Euro-Par, Vol. I*, 1996, pp. 128–135.

[31] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "Mpich-v: toward a scalable fault tolerant mpi for volatile nodes," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002, pp. 1–18.

[32] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems," in *Lecture Notes in Computer Science*. Springer-Verlag, 1999, pp. 533–546.

[33] J. Hursey, J. M. Squyres, and A. Lumsdaine, "A checkpoint and restart service specification for open mpi," Indiana University, Bloomington, Indiana, USA, Tech. Rep. TR635, July 2006.

[34] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," in *The International Journal of Supercomputer Applications*, no. 3, 1991, pp. 63–73. [Online]. Available: citeseer.ist.psu.edu/bailey95nas.html

[35] O. Villa, S. Krishnamoorthy, J. Nieplocha, and D. M. Brown, Jr., "Scalable transparent checkpoint-restart of global address space applications on virtual machines over infiniband," in *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, 2009, pp. 197–206.

[36] R. Brightwell, "A New MPI Implementation for Cray SHMEM," in *EuroPVM/MPI*, 2004, pp. 122–130.

[37] InfiniBand Trade Association, "InfiniBand Architecture Specification, Release 1.2," October 2004.

[38] G. R. Gao and V. Sarkar, "Location consistency-a new memory model and cache consistency protocol," *IEEE Transactions on Computers*, vol. 49, pp. 798–813, 2000.

[39] E. Aprà, A. P. Rendell, R. J. Harrison, V. Tipparaju, W. A. deJong, and S. S. Xantheas, "Liquid Water: Obtaining The Right Answer For The Right Reasons," in *SuperComputing*, 2009.

[40] "Chinook SuperComputer, Environmental Molecular Science Lab, PNNL," http://emsl.pnl.gov.