SAND2013-10002C

Coordination Languages and MPI Perturbation Theory: The FOX Tuple Space Framework for Resilience

Jeremiah Wilke
Scalable Modeling and Analysis
Sandia National Laboratories
Livermore, CA
jjwilke@sandia.gov

***Abstract*—Coordination languages are an established programming model for distributed computing, but have been largely eclipsed by message passing (MPI) in scientific computing. In contrast to MPI, parallel workers never directly communicate, instead "coordinating" indirectly via key-value store puts and gets. Coordination often focuses on program expressiveness, making parallel codes easier to implement. However, coordination also benefits resilience since the key-value store acts as a virtualization layer. Coordination languages (notably Linda) were therefore leading candidates for fault-tolerance in the early '90s. We present the FOX tuple space framework, an extension of Linda ideas focused primarily on transitioning MPI codes to coordination programming. We demonstrate the notion of "MPI Perturbation Theory," showing how MPI codes can be naturally generalized to the tuple-space framework. We also consider details of high-performance interconnects, showing how intelligent use of RDMA hardware allows virtualization with minimal added latency. The framework is shown to be resilient to degradation of individual nodes, automatically rebalancing for minimal performance loss. Future fault-tolerant extensions are discussed.**

I. Introduction

Programming models must always confront a trade-off between imperative and declarative. A more imperative language gives the programmer more flexibility, but more responsibility. Since a declarative language merely prescribes *what* should be computed and not *how*, the compiler or runtime may make non-optimal decisions (or it may make better decisions). No framework is ever purely imperative, though. Even in C the programmer declares a loop without actually implementing every detail. The key question is not either-or, but where programming models should lie on a spectrum between the two. The issue is succinctly summarized by the Alan Perlis quote [1], "A programming language is low level when its programs require attention to the irrelevant."

Here we are primarily concerned with distributed memory parallelism. The most common programming model, message passing (MPI), [2] is essentially imperative. Communicating sequential processes (CSPs) explicitly send messages at a given time to a given place. Resilience often relies on global checkpoints or asynchronous checkpoints with message logging [3], [4]. Other fault-tolerant extensions have been developed or are being proposed for MPI+X [5], but the resilience APIs are still "imperative" in requiring the programmer to specify a fault response. When referring to MPI here, we use it synonymously with the most common usage as bulk-synchronous or send-deterministic. Although it is possible to implement asynchronous task models using MPI_Probe or non-blocking communication, the term MPI here implies an actual programming model, not just a transport layer.

Several alternative programming models are more declarative, including Charm++ [6], Concurrent Collections (CnC) [7], DaGuE [8] or even domain-specific like Uintah [9]. The programmer defines an object interface (chare in Charm++) or workflow (tuple operations in CnC) which describes the work to be done. However, the runtime is still able to make decisions about exactly where and when work is performed. CnC is inspired largely by Linda [10], promoting coordination languages where workers are "decoupled in space and time." Coordination languages such as Linda involve concurrent workers that never directly communicate, instead "coordinating" through a key-value store. Declarative programming models are often implemented as asynchronous task or data-flow execution models. In many cases, the original intent of asynchronous execution was efficiency rather than resiliency. Decoupling workers can hide the latency of global collectives in bulk-synchronous models or simplify load-balancing for irregular computation.

The declarative/imperative spectrum for distributed memory programming is particularly relevant as high-performance computing faces new resilience challenges [11]. For programming models, we must choose an abstract view of the machine that assumes 1) the machine is perfect or 2) the machine can fail. In general, programming models have relied on hardware infallibility, assuming that ECC and analogous schemes will correct all faults. In basic usage, simple C and C++ programs can be "fault-oblivious," assuming perfect hardware. For the next generation of extreme-scale, distributed memory applications (exascale), we must readdress the practical question of how fault-oblivious can the programming model be? MPI can be fault-tolerant, but, by its imperative nature, it is not fault-oblivious. The programmer specifies actions explicitly that may or may not succeed. Declarative models, originally intended to improve coding and execution efficiency, can potentially provide a resilience advantage. With decoupled

workers, a distributed runtime separates when work is declared and when work is executed, providing virtualization. Even if not completely fault-oblivious, decoupled workers can execute fault-amelioration or fault-recovery strategies asynchronously without slowing down the whole computation.

This inherent virtualization of the more declarative coordination languages lent strong interest to fault-tolerant Linda extensions in the 1990s [12]. With renewed emphasis on HPC resilience, we explore here the possibility of a fault-oblivious programming model. We present here the FOX (Fault-Oblivious at eXtreme-scale) tuple framework. In FOX, processes do not perform point-to-point communication with specific ranks, but instead only request data via global identifiers (tuples). Work is driven in a data-flow manner via tasks and data dependencies, increasing efficiency through asynchronous execution.

In last 20 years, Linda and coordination languages have lost out to MPI in scientific computing, which we suggest has two major causes. First, coordination languages can raise communication costs. Processes do not directly communicate and may need to query metadata servers to communicate. MPI rendezvous protocols, however, already require several messages back and forth to negotiate transfers. Even if requiring indirection through a metadata server, communication can match MPI efficiency by intelligent use of hardware primitives. The key-value store frameworks also interfere with data locality. By putting data into a key-value store, data might be migrated off-node that would stay local in MPI. For large arrays, FOX only puts metadata (an RDMA handle) into the key-value store, leaving the large array in place. Both communication latency and data locality are preserved despite indirect coordination through a key-value store. In general, the FOX runtime is designed around data locality, moving work to data rather than data to work. Indeed, going forward, optimal data movement will a critical scheduling concern for HPC.

The second advantage to MPI in scientific computing is the model itself. Tightly-coupled message passing potentially maps more naturally to tightly-coupled physics. Additionally, a runtime may make non-optimal scheduling decisions an imperative programmer could avoid. In the absence of faults, assume there is a hand-coded MPI program that follows an optimal schedule. We argue here that, given simple heuristics and no failures, a simply affinity scheduler can replicate the MPI control flow. The programmer can avoid poor scheduling if she expresses enough information to the scheduler. The FOX framework, following inspiration from Linda and CnC, focuses on *expressiveness*, allowing the programmer to express data partitioning and scheduling hints to the runtime. We coin the term here "MPI perturbation theory", alluding to the famous physics technique. Here the zeroth order problem is a fault-free, perfectly-balanced machine for which an SPMD MPI code is the "solution." Perturbations (faults, load imbalance) are treated as minor corrections to the existing solution.

We therefore argue here that a tuple-space coordination language can provide virtualization without sacrificing efficiency. We demonstrate real-world examples of refactoring an MPI code into tuple-space operations. Excellent performance is observed for medium-sized runs (750cpu) even when certain nodes are degraded. We finish by considering fault-tolerance extensions, discussing how fault-oblivious the framework can be made.

II. Related Work

Coordination languages involve concurrent workers that never directly communicate, instead "coordinating" through a key-value store. In Linda [10], a program is entirely built from three operations: `put`, `rd`, and `in`. An `in` operation reads and removes (takes) the tuple. A simple Linda program with two workers would be

```
Worker 1:
int main()
{
  ...
  put(tsA, "hello", 0);
  ...
}
Worker 2:
int main()
{
  ...
  in(tsA, "hello", i?)
  ...
}
```

The first worker outputs a tuple into space A and the second worker takes the tuple. The `in` operation can return any valid tuple through wildcard operators. A programmer can therefore request any available work rather than a specific task. Common usage is therefore a "bag-of-tasks" work queue. Concurrent collections (CnC) [7] is essentially a subset of Linda with a more extensive runtime. Tasks are enqueued by emitting control tag tuples with dependencies in a data tuple space.

Fault-tolerance extensions were proposed for Linda, based primarily on the use of transactions and replicated state machines. Tuple space transactions could be grouped within atomic guarded statements (AGS) such that either all statements execute or none do [12]. The AGS therefore ensures that failures occurring in the middle of the transaction do not leave the tuple space in a corrupted, intermediate state. Tuple spaces were almost made redundant as replicated state machines [13]. With redundancy, tuple space operations become more expensive, requiring, e.g., atomic multicast to ensure consistency and correctness.

Numerous task frameworks exist independent of coordination programming. DaGuE, originally designed for shared memory, has been extended to distributed memory [8]. Tasks and dependencies are declared via a simple DaGuE syntax with the user explicitly unrolling the DAG gradually at specific points. DaGuE's task framework is similar to FOX, but has not emphasized resilience (to our knowledge).

TASCEL/Scioto are task frameworks that rely heavily on work stealing for load balance [14], [15]. Many resilience features have been explored with TASCEL, in particular ensuring persistence of the task queue with failed nodes and also idempotent correctness of the computation when failed tasks are replayed [16]. While a large degree of asynchrony exists, tasks must be run in collections that are created and closed synchronously.

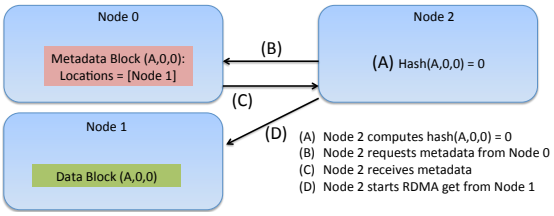

Fig. 1: FOX scheme for reading data in tuple space

Charm++ is essentially an asynchronous task framework, implemented via distributed objects called chares [6]. Data and tasks are delivered to processes as remote object function calls, and the runtime is responsible for load balancing. Resilient extensions have been explored via in-memory checkpointing on buddy processors [17]. Chares also lend them themselves to virtualization in Charm++ since users only specify what actions an object should take but not necessarily where and when. There is strong overlap between the functionality and implementation of Charm++ and the current work, with emphasis on hardware optimizations afforded by state-of-the-art interconnects [18]. The FOX tuple framework differs primarily in the programming model, offering an alternative way to express the parallel decomposition of a program.

Globally visible tuple spaces share some similarity with partitioned global-address space (PGAS) models such as GASNet [19]. APGAS in particular emphasizes asynchronous execution of tasks on "places" distributed across the machine [20]. The recently introduced "global futures" also emphasize remote procedure calls on remote data, asynchronously moving work to data [21].

Many other asynchronous task frameworks exist with emphasis on shared memory or accelerators including Cilk [22], X10 [23], SMPs [24], and StarPU [25]. OpenMP [26] and Intel thread building blocks (TBB) [27] also support some forms of task parallelism. Cilk additionally has a distributed memory version Cilk-NOW [28]. X10 has recently introduced transparent fault tolerance extensions [29].

III. FOX Framework

A. Tuple Operations

Similar to Linda, the FOX framework is built entirely on a limited set of tuple space primitive operations. The application can `put`, `read`, `pull`, or `erase` tuples. Here `pull` is synonymous with the Linda `in` operation. `read` and `pull` functions have the prototype

```
void
read(int ts_id,
     Tuple* t,
     EventListener* listener);
```

Each read operation must specify a tuple space ID, the tuple to be read, and an event listener. In contrast to Linda, there are no blocking operations in FOX. If the requested tuple is found, the function returns and the listener will have a join counter of 0, signaling a successful read. If the requested tuple is not found, the listener object is registered with the tuple space. Each listener must implement an `activate` method. When a matching `put` operation occurs, the listener is activated. This is typically used in task frameworks to signal data dependencies becoming available. In most FOX use cases, however, the event listeners are hidden by the task API.

At present, FOX does not support wildcard tuple reads as done in Linda. Without wildcard support, why do you need to read a tuple if you already know all its elements? First, tuples can synchronize computation. You cannot `read` a tuple unless it has first been `put` into the tuple space. Reading a known tuple can therefore be used as, e.g., a fine-grained barrier.

A known tuple can also be used to read an unknown value. There are two tuple types in FOX: simple tuples and value tuples. Simple tuples are just a collection of elements, all of which are used for labeling the tuple. Value tuples are labeled by a collection of elements, but the tuple stores an unknown value. Consider the tuples:

```
Tuple2<int,int> simple(0,1);
ValueTuple2<int,int,int> value(0,1,2);
```

Both tuples share the label `(0,1)`, but `value` can carry a variable integer. The first two integers in the template refer to the label while the last integer is the value type. In many use cases, the value is the FOX `ArrayData` type, e.g.

```
ArrayData data;
data.instantiate<double>(size);
ValueTuple2<int,int,ArrayData> value(0,1,data);
```

B. Threads

FOX is designed to run with only a single process per physical node. On-node parallelism is provided by explicit use of pThreads. One or more cores is reserved for a remote manager (see below), which constantly polls for incoming tuples. The remaining cores run as task threads. These tasks continually query a node task queue for any ready tasks. In the current version, individual tasks run as single threads and no explicit knowledge of NUMA regions is considered in assigning tasks. For future versions, we plan to tag data not only with the node location but also NUMA region so that scheduling will consider both node and thread affinity. Additionally, we plan to expand the threading model so that data-parallel tasks can be run on multiple threads. This will be particularly important for ensuring that system-level distribution is coarse-grained enough to amortize task overheads while thread-level distribution is sufficiently fine-grained to exploit abundant on-node parallelism.

To ensure thread safety of the tuple space hash tables, each hash table is partitioned into separate (ca. 1000) thread-safe bins. Each bin has its own thread lock for ensuring atomicity of the operations. For a tuple space, many simultaneous operations can occur, but only one operation can occur for a specific tuple at any given time. By mapping into a finite number of bins, we avoid a global tuple space lock and also avoid requiring every tuple to maintain its own lock. The number of bins should be chosen to minimize thread conflicts with reasonable storage requirements.

C. Remote Manager

In distributed versions of Linda, a tuple space can have either local or global scope. FOX follows a similar scheme. Global tuple spaces are implemented as distributed hash tables. Since the number of tuples should be very large, severe load imbalance is unlikely. User-specified hash functions could be incorporated for specific applications to promote locality, e.g. mapping an (X, Y, Z) tuple to a specific processor on a grid.

For large arrays, putting and reading large amounts of data from a distributed hash table is clearly inefficient. Instead of a destination directly reading from a source, data movement is doubled by going through an intermediary. Additionally, the distributed hash table scatters tuples across the machine, potentially putting neighbor data on a far-away node. To correct this, when putting a tuple with `ArrayData`, only the array metadata is actually transferred. Correspondingly, when reading a tuple with `ArrayData`, only the metadata is returned. To fill the array values, the receiving node must use the metadata to perform a remote read, usually in the form of an RDMA get. This scheme is shown in Figure 1. The task API hides all these details (Section IV-B), and most application programmers will never see the details.

Compared to MPI, first reading metadata from a distributed hash table might seem to greatly increase communication latency. However, MPI rendezvous protocols already require several round trips to complete the send (Figure 2). To use RDMA, the source must send a metadata header to negotiate the RDMA transfer and the destination must ack completion. Since FOX runs tasks asynchronously, it effectively overlaps communication and computation, further reducing the overhead of communication latency.

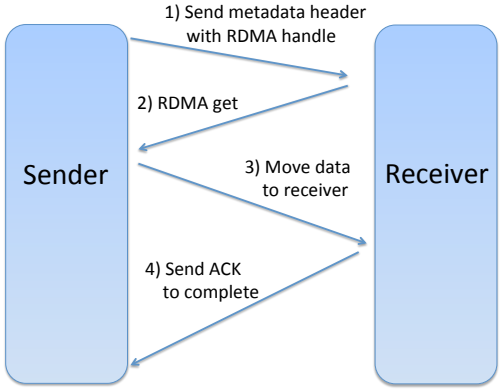

Fig. 2: RDMA rendezvous protocol for MPI_Send/MPI_Recv Pair

IV. Systolic Matrix-Matrix Multiplication

We initially explore the FOX tuple framework in the context of matrix-matrix multiplication.

A. Basic MPI Algorithm

We show here one version of 2D systolic matrix-matrix multiplication (Figure 3) for computing $C = AB$. For simplicity, we only consider square matrices. The matrices are partitioned into an $N \times N$ grid. For each block C_{ij}, it must accumulate N contributions

$$C_{ij} = \sum_k A_{ik} B_{kj} \tag{1}$$

On the first iteration, each product block accumulates a unique contribution, e.g.

$$\begin{aligned} C_{00} &+= A_{00} B_{00} \\ C_{01} &+= A_{01} B_{11} \\ &\vdots \end{aligned} \tag{2}$$

Moving to the next iteration, the blocks of A and B shift as a systolic array (Figure 3) and each product block again accumulates a unique contribution.

$$\begin{aligned} C_{00} &+= A_{01} B_{10} \\ C_{01} &+= A_{02} B_{21} \\ &\vdots \end{aligned} \tag{3}$$

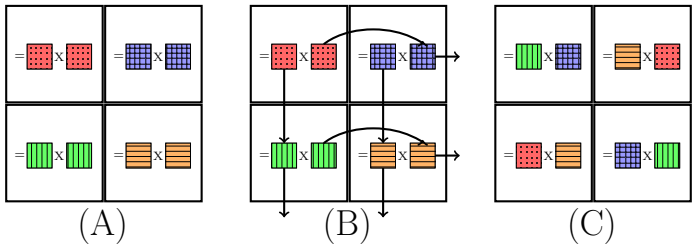

Fig. 3: Basic systolic matrix multiplication $\mathbb{C} = \mathbb{A}\mathbb{B}$ for 2x2 matrices. The blocks of $\mathbb{C}$ (large boxes) accumulate contributions from $\mathbb{A} \times \mathbb{B}$ (small boxes). **(A)** The initial distribution of matrix blocks in the first iteration. **(B)** The movement of matrix blocks after the first iteration. **(C)** The distribution of matrix blocks on the second iteration.

This algorithm is highly synchronous, making it initially seem a bad choice for an asynchronous task runtime. However, shown below, we are still able to construct a sensible task decomposition in the FOX tuple framework. The MPI code is shown schematically.

```
for (int iter=0; iter < niter; ++iter){
    MPI_Isend(myBlockA,...);
    MPI_Isend(myBlockB,...);
    MPI_Irecv(nextBlockA,...);
    MPI_Irecv(nextBlockB,...);

   /** Matrix multiplication work */

   MPI_Wait(...);
   myBlockA = nextBlockA;
   myBlockB = nextBlockB;
}
```

To make the performance comparison between FOX and MPI "fairer," we prefetch each iteration to try and overlap communication and computation as much as possible in the MPI code.

B. FOX Implementation

We here introduce a FOX project for matrix-multiplication. In particular, we try to replicate systolic matrix multiplication within the task-driven framework. Every FOX code has a well-defined structure, containing minimally these files:

Header	Source
declare_deps.h	define_deps.cc
declare_tasks.h	define_tasks.cc

File names are self-explanatory. First, one must declare the the data types (dependencies) that tasks will use.

```
#include <libfoxy/fox.h>
fox_dependency_declare(
    right_matrix_block, // Dependency name
    char, // Matrix label
    int, // Iteration number
    int, // Row index
    int, // Column index
    ArrayData // FOX class for arrays);
fox_dependency_declare(left_matrix_block, ...);
fox_dependency_declare(product_matrix_block, ...);
```

Underneath the hood, FOX uses macros and C++ templates to generate the code. Any type mismatches between tasks and dependencies are compile-time errors. All macros and functions are included via `fox.h`.

Next we must declare the tasks.

```
#include "declare_deps.h"
fox_tuple_task_declare(
    multiply, // Name of task
    int, // Iteration number
    fox_dependency(product_matrix_block),
    fox_dependency(left_matrix_block),
    fox_dependency(right_matrix_block) );
fox_init_declare(sysmxm_params);
```

Since we are doing matrix multiplication, each task requires three matrix blocks. The names were declared in `declare_deps.h` and must be wrapped in the `fox_dependency` macro. Additionally, we declare a task `init` that takes as input type `sysmxm_params` (not shown). The task is declared as a broadcast. When `init` is invoked, it will actually generate many tasks, unrolling the DAG necessary to execute a broadcast collective. The macros implicitly declare functions `multiply_fxn` and `init_fxn` which must be implemented for the task (see below).

Once tasks and dependencies are declared, they must have their symbols defined for linking. For convenience, we include all headers via a `project.h`

```
/** define_deps.cc */
#include "project.h"
fox_dependency_define(left_matrix_block);
fox_dependency_define(right_matrix_block);
fox_dependency_define(product_matrix_block);

/** define_tasks.cc */
#include "project.h"
fox_task_define(multiply);
fox_init_define();
```

Once declarations and definitions are done, the actual main routine can be implemented.

```
int main()
{
    FoxRuntime::init();
    fox_append_reduce_dependency(
      multiply,product_matrix_block);

    fox_append_migrate_dependency(
      multiply,left_matrix_block);

    fox_append_migrate_dependency(
      multiply,right_matrix_block);

    fox_initial_bcast(&get_params);

    FoxRuntime::run();
    return 0;
}
```

After the runtime is initialized, dependencies must be appended to the multiply task. Even though they are declared in the header file, we currently require them to be explicitly registered in the code. Many dependencies will be declared as simple `read` dependencies. For the systolic matrix-matrix multiplication $\mathbb{C} = \mathbb{A}\mathbb{B}$, however, more complicated dependency actions are required. The $\mathbb{A}$ and $\mathbb{B}$ blocks move such that on each iteration, a different node "owns" the matrix block. This pattern we call a `migrate` dependency. A `migrate` dependency initially performs the actions of a `pull`. However, after the `pull` completes, the array on the source node is deleted and the reader is registered as the new owner. The flow of operations is shown schematically in Figure 4. The user does not manage these details, however. The dependency is just declared as a `migrate` dependency and the FOX runtime performs all the operations.

On each iteration, the product block accumulates a new contribution. It is therefore both an input (reading the old values) and output (accumulating new values). It must also be atomic, accumulating only one contribution at a time to avoid inconsistencies. This dependency can be declared as a `reduce` dependency and the FOX runtime will manage all the read and atomic locking operations. Both `migrate` and `reduce` actually have the same implementation in the current version of FOX. When the tuple metadata is pulled, it is no longer visible to other nodes, making the next operation atomic.

The labeling of dependencies holds some similarities with parallel algorithmic skeletons [30]. Common parallel data movement patterns are encapsulated as library functions (or macros, in this case) implemented entirely on tuple space operations. We are therefore free to implement as much "template" functionality in the library, and all fault tolerance techniques developed for tuple space frameworks automatically apply.

Once all dependencies are appended to tasks, we start the first set of tasks by calling `fox_initial_bcast`. The FOX runtime chooses the root of the broadcast. The broadcast takes a function pointer as argument, which is conditionally invoked on the chosen root process to read parameters (presumably from some input file). Once the broadcast tasks have been declared, the FOX runtime is started.

Starting work through an initial broadcast is designed to ease the transition from existing MPI codes. In a `init.cc` file, the user must implement the init function.

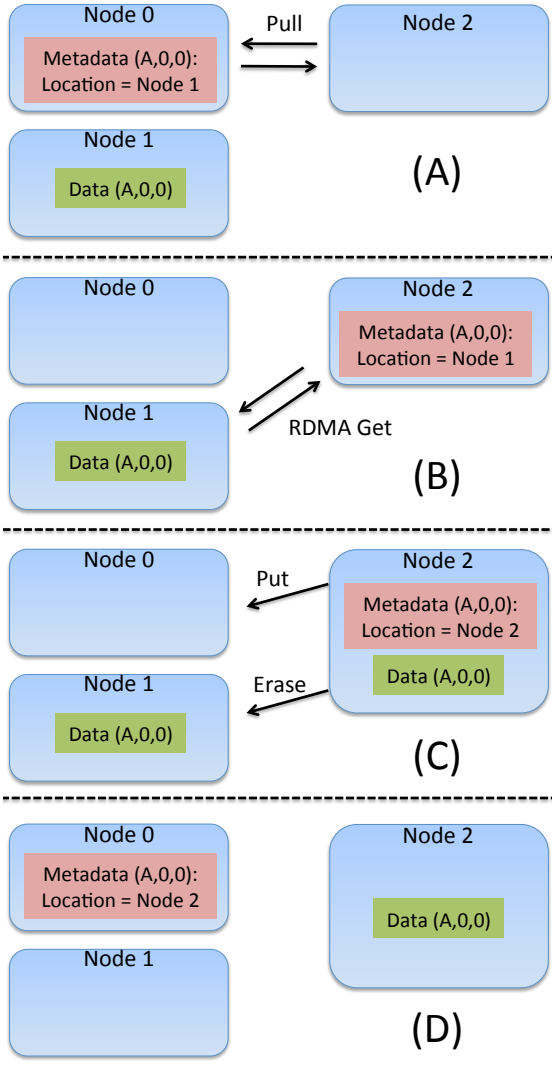

Fig. 4: Migrate dependency tuple actions. A) The tuple metadata is pulled from Node 0. The tuple is no longer globally visible. B) An RDMA get transfers data from Node 1 to Node 2. C) Tuple metadata is put back with the updated location. The original location is erased. D) The data and location metadata now refer to Node 2.

```
void
init_fxn(int me, int nworker,
  const sysmxm_params& bcast_params)
{
...
}
```

The function prototype must include two extra integer parameters `me` and `nworker`, which are essentially synonymous with `MPI_Comm_rank` and `MPI_Comm_size`. In most use cases, work should be over-decomposed, generating many tasks per process. FOX allows the level of over decomposition to be specified, potentially generating many virtual ranks per process. This bears some similarity to adaptive MPI (AMPI) [31]. However, the notion of rank only applies to that single task (function). Once a broadcast is received with parameters, the initialize task can run essentially assuming it is MPI process with rank `me` out of `nworker` processes. This defines an initial partitioning of the work.

First, the initialization loops through its subset of matrix blocks (shown in pseudocode) and then allocates and initializes the data.

```
int first_iter = 0;
for (row, link) in my left matrix subset  {
    ArrayData data;
    data.instantiate<double>(blocksize);
    fox_put_migrate_dependency(
        left_matrix_block,
        'A', first_iter, row, link, data);
}
```

The list of arguments passed to the put dependency function must match those declared in `declare_deps.h`. Type matching is a compile-time check through the C++ template system. The FOX runtime creates a tuple that describes the matrix block and contains all necessary metadata. This tuple is then put into the `left_matrix_block` tuple space. If any `read` or `pull` operations are waiting on the tuple, their event listeners will be signaled letting them know the tuple has been put. Similar code is used for the right matrix blocks. For subtle reasons, dependencies must be labeled by iteration number. This would not be necessary if declared as a simple read dependency, but is required for migrate dependencies.

For the product matrix blocks, we similarly run

```
for (row, col) in my product matrix subset  {
    ArrayData data;
    data.instantiate<double>(blocksize);
    fox_put_reduce_dependency(
        product_matrix_block,
        'C', row, col, data);
}
```

Once the matrix blocks are created, we can create the initial set of tasks for the first iteration. One could theoretically create all tasks initially, but (as shown later), we choose to gradually unroll the task graph iteration-by-iteration.

```
int first_iter = 0;
for (row, col) in my product matrix subset  {
    int link = (row+col) % nblocks;
    fox_dependency(left_matrix_block)
        left('A', iter, row, link);

    fox_dependency(right_matrix_block)
        right('B', iter, link, col);

    fox_dependency(product_matrix_block)
        product('C', row, col);

    fox_compute(multiply,
     first_iter, product, left, right);
}
```

Here we choose the appropriate dependencies for each product block. We then create a `multiply` task by calling `fox_compute`. This completes initialization, creating the necessary data (input dependencies) and first task. The FOX runtime can now begin scheduling multiplication tasks. The task descriptor is just a tuple (or tuple of tuples). The tuple is `put` into a task tuple space. As in CnC, the `put` operation spawns a task.

To implement the multiply task, the user must implement `multiply_fxn` with arguments matching those declared in `declare_deps.h`.

```
void multiply_fxn(
 int iteration,
 fox_dependency(product_matrix_block)& prod_tuple,
 fox_dependency(left_matrix_block)& left_tuple,
 fox_dependency(right_matrix_block)& right_tuple)
{
  ...
}
```

Upon entering the function, to actually execute the multiplication task, the dependency tuples must be unpacked.

```
char l_label = left_tuple.first();
int iter = left_tuple.second();
int row = left_tuple.third();
int link = left_tuple.fourth();
ArrayData& arr_data_left = left.value();
double* left_data = (double*) arr_data_left;
```

Before doing the matrix multiplication work, we further unroll the task graph

```
int next_iter = iteration + 1;
if (next_iter < max_iter){
  int next_link = (link+1) % nblocks;
  fox_dependency(left_matrix_block)
     next_left('A', next_iter, row, next_link);

  fox_dependency(right_matrix_block)
      next_right('B', next_iter, next_link, col);

  fox_compute(multiply,
      next_iter, product, next_left, next_right);
}
```

After unpacking the tuples to get the `double*` arrays, we can perform the matrix multiplication work.

To terminate the code, FOX provides a terminate function to signal the runtime that all tasks have been completed. This essentially amounts to a binary tree barrier with each tree node its own task. For more complicated applications, more involved termination detection would be necessary, usually involving voting schemes on a spanning tree [32], [33].

```
/** If the last iteration,
   call terminate for
   this product block */
if (next_iter == max_iter){
    int idx = row * ncols + col;
    fox_terminate(idx, nblocks);
}
```

We must separately signal termination for all blocks. `fox_terminate` therefore receives two parameters: the index of the current block and the total number of blocks. Once all product blocks have signaled termination, the FOX runtime broadcasts a terminate signal to all nodes and the code finishes.

For 3x3 matrix multiplication, the generated task graph is shown in Figure 5. Although the graph is structured in levels by iterations, many opportunities for asynchronous execution exist. The computation terminates by each matrix block passing votes up a termination tree.

C. MPI Perturbation Theory

In the MPI algorithm outlined in IV-A, product blocks remained local while the multiplying blocks were communicated in a systolic array. Following the idea of "MPI perturbation theory", given a simple scheduling heuristic, the FOX scheduler should replicate MPI in a fault-free environment. By default, FOX uses a simple affinity scheduler, sending tasks where the most dependencies are local (measured in number of bytes). To replicate MPI in computing $\mathbb{C} = \mathbb{AB}$, the scheduler should give preference to keep product blocks ($\mathbb{C}$) local. The simplest way to do this in FOX is a slight modification of the dependency registration.

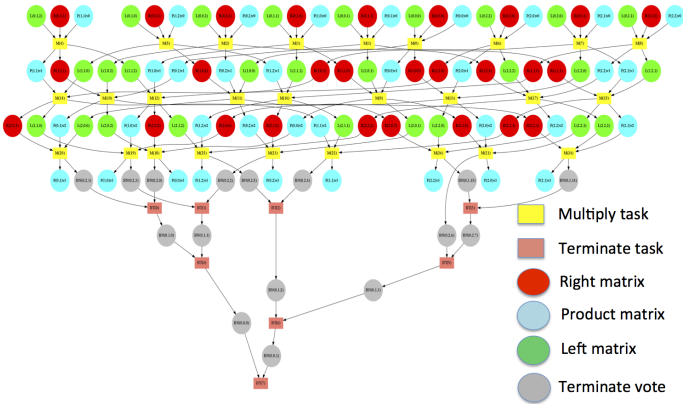

Fig. 5: The generated task graph for matrix multiplication showing multiply and termination tasks.

```
fox_append_weighted_reduce_dependency(
    multiply, product_matrix_block, 3);

fox_append_migrate_dependency(
    multiply,left_matrix_block);

fox_append_migrate_dependency(
    multiply,right_matrix_block);
```

This assigns a heavier affinity for product blocks to the scheduler. When scheduling tasks, assuming no load imbalance develops, the FOX scheduler will exactly recreate the systolic array algorithm, keeping product blocks local and migrating the input matrix blocks. The use of `migrate` dependencies with weighting factors is a weaker version of persistent load balancing [34]. After a product block is migrated to another node, it is unlikely to be migrated again. When a chunk of work is stolen, unless severe load imbalance persists, it remains stolen across iterations. The thief (semi)-permanently keeps work taken from the victim.

When faults occur, either from a node completely failing or a node degrading in performance due to, e.g., overheating, work must be rebalanced. The FOX task manager automatically checks load balance through a "heartbeat." At regular periods, each node sends a tuple to its neighbors with load balance information. By default, the load balance information is a measure of thread idle time. If a node sees an idle neighbor, it shares work, pushing tasks from its queue to the neighbor (rather than work stealing in the opposite direction). Custom logger classes can be used to make work sharing decisions more application-specific. In the current work, we have also tested an "iteration logger" that tracks the average iteration number. If a node is working on an iteration much lower than its neighbors, it will share work. The logger object can be easily configured to send an arbitrary tuple on each

heartbeat. Physically meaningful information can therefore be used in work sharing. Although not done here, a node could share the most useful work based on physical locality on a grid learned from the tuple.

V. Experimental Setup

Tests were run on Hopper, a Cray XE6 system at the National Energy Research Scientific Computing Center (NERSC) [35]. The Cray XE6 consists of dual-socket AMD Magny-Cours compute nodes with a Gemini interconnect. Each socket contains two NUMA regions (dies) with six cores each for a total of 24 cores per node. All NUMA regions are connected by HyperTransport links. The network interface controller (NIC) is attached to the first die and all other dies must traverse the HyperTransport links to reach the NIC. Limited tests were run (192-768 cpu; 8-32 nodes). To simplify implementation, MPI tests required a square number of processors. One FOX process ran per socket with ten task threads and two communication threads per process. MPI was run both with one MPI process per core and also with one process per NUMA region with OpenMP.

All communication in FOX directly used the Cray GNI API for communication [36], [37]. On initialization, short message (SMSG) mailboxes were created on each node. The SMSG mailboxes allow single-trip, low-latency sends for small (<512B) messages, which is optimal for sending many small tuples. Large messages used RDMA gets, for which RDMA descriptors were obtained from tuple reads. The SMSG sends use a fast-memory access (FMA) pathway on the NIC while RDMA transfers use a separate block-transfer engine (BTE).

VI. Results

We first compare performance of MPI to FOX (Figure 6). In general, performance is very similar, being largely determined by computation. As mentioned above, the MPI code does overlap communication and computation. For 192 cpu, however, FOX performs better, suggesting the asynchronous task framework achieves a better overlap of communication and computation.

MPI is running 24 worker threads per node, which occasionally pause to complete communication. FOX, however, only runs 20 worker threads per node, dedicating four threads per node to communication. Despite the extra workers, MPI does not show an appreciable speedup. Likely this is due to memory contention. 20 workers already saturates the memory and 24 workers provides little improvement.

To test how well the task framework can rebalance in the presence of faults, we tested the performance of FOX when one of the nodes becomes degraded. This was achieved by simply repeating tasks N times, resulting in an N-fold slowdown. Using only the default load balancer based on idle time, FOX achieved mostly favorable results (Figure 7). For a 2x slowdown, almost no performance degradation is observed. Compared to MPI, which cannot rebalance, performance is much better. For a 4x slowdown, little performance loss is seen at 384 cpu. However, at 792 cpu, performance

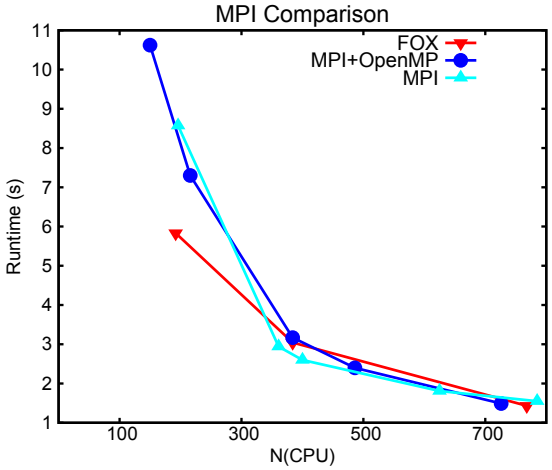

Fig. 6: Comparison of FOX matrix-multiplication performance to MPI

drops significantly. Increasing the number of nodes lowers the number of tasks per node. For 384 cpu, the problem is still sufficiently over-decomposed to allow efficient rebalancing of work. At 792 cpu, there are not enough tasks for the default load balancer to intelligently rebalance.

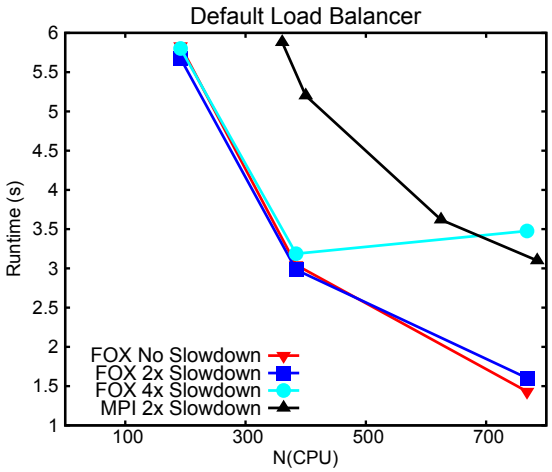

Fig. 7: Performance of FOX matrix-multiplication with degraded nodes using default load balancer.

The degraded node experiment was repeated using a custom logger (Figure 8). Load balance decisions were now based on the average iteration number of the tasks. The degraded node quickly falls behind and sees that its neighbors are working on the next iteration. For a 2x slowdown, results are very similar to the default load balancer. For a 4x slowdown, performance is still excellent for 384 cpu. The iteration load balancer performs better than the default load balancer for 768 cpu, but is still unable to maintain the scaling.

VII. Fault-Oblivious Extensions

We explore a few extensions for the fail-stop case in which a node is completely lost, along with all tuple space data stored on it. Some form of replication is therefore required to ensure

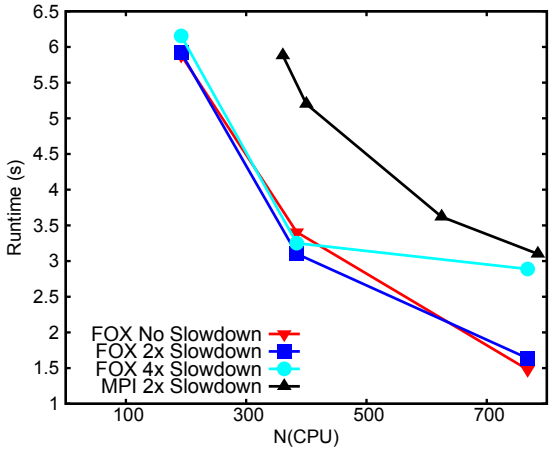

Fig. 8: Performance of FOX matrix-multiplication with degraded nodes using custom load balancer.

durability of the tuple space and array data. As mentioned previously, FOX presents a task interface with specific dependency types, but is constructed through tuple space operations. Fault-tolerance therefore only needs to address tuple space primitive operations, independent of what functionality is built on top.

A. Tuple replication

The distributed hash table can easily be extended for replicas, generalizing the hash function to generate N replicas for each tuple. The primary concern when using replicas is consistency and correctness. Program execution must be *linearizable* [13]. Any parallel execution must be equivalent to some serialized list of sequential operations. Two general schemes are possible for replicating operations. First, a tuple space operation is delivered to all replicas. Some voting procedure is applied to ensure consistency and the elected tuple is either written or returned. In the second method, a primary replica is appointed. Tuple space operations are broadcast to all replicas, but the primary mediates all operations to ensure consistency. If the primary fails, one of the backups is promoted. We are pursuing the second approach. Since tuple spaces are already DHTs, load is distributed across the entire machine. Additionally, because simple hash functions are used to map tuples, electing new primaries can by implemented by simply shifting the original hash.

B. Transactions

The most difficult fault-tolerance aspect is implementing all-or-nothing transactions. As mentioned in the introduction, methods based on atomic guarded statements (AGS) were previously proposed [12] In FOX, data movement operations (such as `migrate`) involve a sequence of tuple operations that represent, in sum, a single transaction. If any step fails, all steps should be reversed to avoid corrupted, intermediate state. Read operations do not need to be tracked in transactions, but `put` and `pull` operations must be made "reversible." In general, these operations must be tagged as transactions and be aware of all participants. Until the transaction is acknowledged, the new tuple state cannot be committed. If a failure notification for a given node is received, all transactions involving that node must be canceled. Because common parallel operations are encapsulated in FOX library calls, most users will not be troubled with transaction details.

C. Data replication

Different data resilience strategies are all compatible with FOX, including persistent memory (NVRAM), in-memory checkpoint, or even parity schemes. When `ArrayData` types are `put` into the tuple space, a resilience callback must be activated to back up the array. Because FOX emphasizes heavily asynchronous execution, the latency of creating backups can be at least partially hidden. On failure, the backup copy must be used to restore the array data. As shown above, techniques exist for making the tuple space itself resilient. Suppose we create an array labeled as `Tuple(0,1,2)` referencing a block of X, Y, Z grid points. A backup copy could be written to NVRAM and the location stored in the tuple space, with the modified label `Tuple("nvram",0,1,2)`. If the primary copy fails, the NVRAM backup can be retrieved from the tuple space.

D. Work rebalancing

When a node fails, three options exist for recovery. The node can be rebooted and reintegrated. Alternatively, a spare node could be swapped in. Finally, the node could simply be removed and work rebalanced across the remaining workers. Each strategy comes with certain cost tradeoffs which will need to be explored. While rebooting or swapping, the new node will not be making any progress on its tasks. However, the FOX runtime system may hide this problem via load balancing. When the node launches, depending on the heartbeat mechanism, the runtime may recognize the node is behind its neighbors and migrate work. When the node catches up, work will migrate back.

VIII. Conclusions

We have presented the FOX tuple framework. Users interact with a simple API for expressing tasks, dependencies, and common parallel data movement patterns. Underneath, a C++ template system maps onto standard tuple space operations from Linda. As HPC faces new resilience challenges in the future, we are trying to maximally reuse established solutions from the past. By constructing our task runtime on tuple operations, we can utilize the large body of fault-tolerance research that exists for tuple coordination languages. We have tried to correct two important shortcomings relative to MPI: communication costs and expressiveness. The FOX framework matches MPI performance in the absence of faults and exhibits almost no performance drop when a single node is degraded. Future work will focus on larger scales and more applications.

IX. Acknowledgments

This work was supported by the U.S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) Advanced Simulation and Computing program and the DOE Office of Science Advanced Scientific Computing Research program. SNL is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the DOE NNSA under contract DE-AC04-94AL85000.

References

[1] A. J. Perlis. Special Feature: Epigrams on Programming. *SIGPLAN Notices* 17, 7 (1982).

[2] M. P. I. Forum. *MPI: A Message-Passing Interface Standard: Version 2.1*. 2008.

[3] A. Bouteiller, T. Ropars, et al. Reasons for a Pessimistic Or Optimistic Message Logging Protocol in MPI Uncoordinated Failure, Recovery. *CLUSTER 2009: IEEE International Conference on Cluster Computing*, pp. 1, 2009.

[4] A. Guermouche, T. Ropars, et al. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications. *IPDPS '11: 25th International on Parallel and Distributed Processing Symposium*, pp. 989, 2011.

[5] W. Bland, A. Bouteiller, et al., 'An Evaluation of User-Level Failure Mitigation Support in MPI' in *Recent Advances in the Message Passing Interface* edited by J. Träff, S. Benkner, J. Dongarra. (Springer Berlin Heidelberg, , 2012).

[6] L. V. Kale, S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. *OOPSLA 1993: 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 91, 1993.

[7] M. G. Burke, K. Knobe, et al. The Concurrent Collections Programming Model. *Technical Rep. TR 10 12* , (2010).

[8] G. Bosilca, A. Bouteiller, et al. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. *Parallel Comput.* 38, 37 (2012).

[9] M. Berzins, J. Schmidt, et al. Past, Present and Future Scalability of the Uintah Software. *Extreme Scaling Workshop*, pp. 1, 2012.

[10] N. J. Carriero, D. Gelernter, et al. The Linda Alternative to Message-Passing Systems. *Parallel Comput.* 20, 633 (1994).

[11] M. Snir, R. W. Wisniewski, et al.. *Addressing Failures in Exascale Computing*. 2013.

[12] D. E. Bakken, R. D. Schlichting. Supporting Fault-Tolerant Parallel Programming in Linda. *IEEE Transactions on Parallel Distrib. Syst.* 6, 287 (1995).

[13] A. Xu, B. Liskov. A Design for a Fault-Tolerant, Distributed Implementation of Linda. *FTCS: 19th International Symposium on Fault-Tolerant Computing*, pp. 199, 1989.

[14] J. Dinan, D. B. Larkins, et al. Scalable Work Stealing. *SC '09: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2009.

[15] J. Dinan, S. Krishnamoorthy, et al. Scioto: A Framework for Global-View Task Parallelism. *ICPP 2008: 37th International Conference on Parallel Processing*, pp. 586, 2008.

[16] W. Ma, S. Krishnamoorthy. Data-Driven Fault Tolerance for Work Stealing Computations. *ICS 2012: 26th ACM International Conference on Supercomputing*, pp. 79, 2012.

[17] Z. Gengbin, N. Xiang, et al. A Scalable Double in-Memory Checkpoint and Restart Scheme Towards Exascale. *FTXS 2012: Workshop on Fault Tolerance for HPC at Extreme Scale*, pp. 6, 2012.

[18] S. Kumar, Y. Sun, et al. Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q. *IPDPS 2013: 27th International Parallel and Distributed Processing Symposium*, pp. 689, 2013.

[19] D. Bonachea. GASNet Specification, v1.1 U.C. Berkeley Tech Report (UCB/CSD-02-1207). , (2002).

[20] V. Saraswat, G. Almasi, et al. The Asynchronous Partitioned Global Address Space Model. *AMP '10: First Workshop on Advances in Message Passing*, 2010.

[21] D. Chavarria-Miranda, S. Krishnamoorthy, et al. Global futures: a multi-threaded execution model for global arrays-based applications. *CCGrid '12: 12th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 393, 2012.

[22] R. D. Blumofe, C. F. Joerg, et al. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Notices* 30, 207 (1995).

[23] P. Charles, C. Grothoff, et al. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. *OOPSLA 2005: 20th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 519, 2005.

[24] J. M. Perez, R. M. Badia, et al. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. *CLUSTER 2008: IEEE International Conference on Cluster Computing*, pp. 142, 2008.

[25] C. Augonnet, S. Thibault, et al. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency Comput. Practice Exp.* 23, 187 (2011).

[26] B. Chapman, G. Jost, et al., *Using OpenMP: Portable Shared Memory Parallel Programming*. edited by W. Gropp, E. Lusk. MIT Press, Cambridge, MA (2008).

[27] J. Reinders, *Intel Threading Building Blocks*. O'Reilly, (2007).

[28] R. D. Blumofe, P. A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. *USENIX ATC '17: USENIX Annual Technical Conference*, 1997.

[29] C. Xie, Z. Hao, et al. X10-FT: Transparent Fault Tolerance for APGAS Language and Runtime. *PMAM 2013: International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 11, 2013.

[30] R. Marques, H. Paulino, et al. Algorithmic Skeleton Framework for the Orchestration of GPU Computations. *ICPP '13: 19th International Conference on Parallel Processing*, pp. 874, 2013.

[31] C. Huang, G. Zheng, et al. Performance Evaluation of Adaptive MPI. *PPoPP '06: 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 12, 2006.

[32] N. Francez, M. Rodeh. Achieving Distributed Termination without Freezing. *IEEE Transactions on Software Engineering* SE-8, 287 (1982).

[33] J. Lifflander, P. Miller, et al. Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection. *PPoPP '13: 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 13, 2013.

[34] J. Lifflander, S. Krishnamoorthy, et al. Work Stealing and Persistence-Based Load Balancers for Iterative Overdecomposed Applications. *HPDC '12: 21st International Symposium on High-Performance Parallel and Distributed Computing*, pp. 137, 2012.

[35] H. Shan, N. J. Wright, et al. A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI. *PMBS '11: 2nd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, pp. 13, 2011.

[36] R. Alverson, D. Roweth, et al. The Gemini System Interconnect. *HOTI '10: 18th Annual Symposium on High Performance Interconnects*, pp. 83, 2010.

[37] S. Yanhua, Z. Gengbin, et al. A uGNI-based Asynchronous Message-Driven Runtime System for Cray Supercomputers with Gemini Interconnect. *IPDPS '12: 26th International Parallel and Distributed Processing Symposium*, pp. 751, 2012.