Tuple Switching Network – When Slower May Be Better

Justin Y. Shi, Moussa Taifi, Abdallah Khreishah, and Jie Wu

Department of Computer & Information Sciences
Temple University
Philadelphia, PA 19122
{*shi,moussa.taifi,akhreish,jiewu*}*@temple.edu*

Abstract

This paper reports an application dependent network design for extreme scale high performance computing (HPC) applications. Traditional scalable network designs focus on fast point-to-point transmission of generic data packets. The proposed network focuses on the sustainability of high performance computing applications by statistical multiplexing of semantic data objects. For HPC applications using data-driven parallel processing, a tuple is a semantic object. We report the design and implementation of a tuple switching network for data parallel HPC applications in order to gain performance and reliability at the same time when adding computing and communication resources. We describe a sustainability model and a simple computational experiment to demonstrate extreme scale application's sustainability with decreasing system mean time between failures (MTBF). Assuming three times slowdown of statistical multiplexing and 35% time loss per checkpoint, a two-tier tuple switching framework would produce sustained performance and energy savings for extreme scale HPC application using more than 1024 processors or less than 6 hour MTBF. Higher processor counts or higher checkpoint overheads accelerate the benefits.

Keywords:
Application dependent networking, Sustainable high performance computing.

1. Introduction

Traditional scalable network designs focus on the fast transfer of generic data packets. In theory, packet switched data networks are sustainable and scalable since the packet transmission performance and reliability improve as we add networking components. Unfortunately, higher level applications rely on application dependent communicating objects for their tasks. Except for applications that can be easily partitioned into independent parallel tasks [1], the semantic differences between the application's data objects and the low level data packets make it difficult for higher level applications to gain the full benefits of data network sustainability.

Unlike packet switched data networks, HPC program interfaces (API) handle only two states of each transmission: success or failure. The timeout (unknown) state is left to the programmer's own devices. Due to the lack of alternatives, timeout is treated identical to failure. Since the probability of transient failures grows proportionally to the component count in the processing environment, from the applications' perspective, every transient error is a single point failure. Any transient failure can bring the entire application to a complete halt. Restarting the application from the last failed point requires periodical checkpointing. Even with the optimal checkpoint intervals, work (and energy) loss is inevitable. Bigger losses are expected for larger applications. These problems are most pronounced in HPC applications. As we continue to grow the component counts in supercomputers and high performance clusters, the two-state-only timeout practice ensures continued decline of application's mean time between failures (MTBF) ([2], [3]).

For HPC applications, it is commonly accepted that to achieve higher performance, reliability must be sacrificed. For higher application reliability, performance must be sacrificed [4]. Removing this limitation is generally considered very difficult.

Our research has found that sustainable solutions for large systems are feasible if tackled by application types [5]. The idea is to apply statistical multiplexing on application dependent objects, if we can define them correctly. For HPC applications, the first challenge is to identify the unit of transmission that would be acceptable for all application types. Unlike traditional HPC interconnection networks, a tuple switching network's unit of transmission is an application dependent data tuple. Since data parallel processing is a well established discipline ([6],[7] and [8]), using the data tuple as the unit of transmission should be theoretically acceptable for all HPC application types. The next challenge is to design a parallel processing paradigm and architecture that can support statistical multiplexing of user data processing and transmission leveraging multiple redundant computing nodes and networks. By the End-to-End principle [9], the HPC application's API must also be revised to include the statistical multiplexing semantics.

This paper reports the architecture of a tuple switching network, its processing paradigm and implementation details. Tuple-driven parallel processing decouples HPC programs in time and in space thus allowing statistical multiplexing of communicating tuples. Like the packet switched data networks, the tuple switching network statistically mitigates the unknown states of tuple transmission and processing by leveraging potentially massive redundant networks and computing nodes. If we consider the packet "store-and-forward" equivalent to an HPC application's checkpoint, the tuple switching network is identical in structure to a packet switching network [10]. Thus, given properly partitioned tuple (data) parallel programs, a tuple switching network could afford high sustainability for extreme scale HPC applications leveraging virtually unlimited number of redundant networks and computing nodes.

Tuple switching network marks a departing point from peak performance-only HPC designs. Since packet processing is done "statelessly", the processing paradigm is essentially a stateless parallel processing (SPP) machine. Tuple switching network supports both homogeneous and heterogeneous processors and networking technologies. Therefore it encourages continued diverse developments of lower level processors and networks including on-chip networks, many-core and multi-core processors and multiple point-to-point interconnects and system-wide networks. A two-tier tuple switching network will be capable of leveraging legacy HPC applications for future fault tolerant extreme scale applications. To strengthen our arguments, we also include a mathematical sustainability model and a simple computational experiment to demonstrate the expected application performance in extreme scales of rapidly decreasing MTBFs. Our results show that assuming three times slowdown of the statistical multiplexing layer, for an application using 1024 processors with 35% time loss per checkpoint, the two-tier tuple switched network will produce sustained performance and energy savings for systems with less than 6 hour MTBF. Higher processor counts or higher checkpoint overheads accelerate the benefits.

Since sustainable systems must deal with component faults – a long standing non-trivial topic, to contain the complexity of discussion we focus on the sustainability of HPC applications in general. Programming examples are included in Appendix A. This paper is organized as follows: Section 2 is a survey of parallel processing paradigms and their performance and fault tolerance methodologies. We focus on their intrinsic features for applications sustainability. Section 3 discusses the motivation and the concept of semantic statistical multiplexing. Section 4 presents tuple-driven parallel processing paradigm. Section 5 describes the architecture of the proposed tuple switching network, its implementation and its sustainability model. Section 6 reports the design and implementation of a simple computational experiment in order to validate the sustainability model with component failures. Section 7 contains discussion of the results and their relevance to established HPC practices and benchmarks. Section 8 contains the summary and future directions.

2. HPC Programming Paradigms and Application Fault Tolerance

2.1. HPC Programming Paradigms

From programming style's perspective, there are two HPC programming paradigms: explicit and implicit. Explicit parallel programming requires the programmer to specify in detail how concurrent processes communicate with one another. This allows the application to take advantage of processing architecture's features for higher performance. There are two popular process-to-process communication methods: message passing (MPI) [11] and shared memory (OpenMP) [12]. Detailed instructions are expected by the explicit parallel processing system for executing the parallel programs.

In the last two decades, explicit parallel programs have demonstrated very high performances. They have become the international standards for supercomputer benchmarks (http://www.top500.org). For extreme scale applications, however, we still face the following challenges:

- Programming complexity is too high. An application programmer is required to be a domain knowledge expert, familiar with supercomputer hardware details and a skilled coder.
- Increasing failure rate. As supercomputers grow in socket counts, the application mean time between failure (MTBF) is fast shrinking. The explicit programming requirements make failure prevention for large scale applications difficult.
- Energy efficiency. Increasing failure rates force the applications to frequent checkpoint and restarts (CPR). For extreme scale applications, even with the "greenest" processor designs, the CPR overheads would be cost prohibitive.

Implicit parallel programming hides the process-to-process interaction from the programmer. The programmer only specifies the rules for data generation and consumption. The runtime system is responsible for the interactions between programs and data. Changing processing granularity involves sending out different number of working assignments. No recoding should be required for granularity tuning unless the tuning range needs to cross the iteration boundaries.

To date, there are approximately five different implicit parallel programming paradigm implementations:

- Hardware-based. The very first data parallel machines, such as the MIT Tagged Token machine [6] and the Manchester dataflow machine [13], used hardware data matching circuits to activate programs.

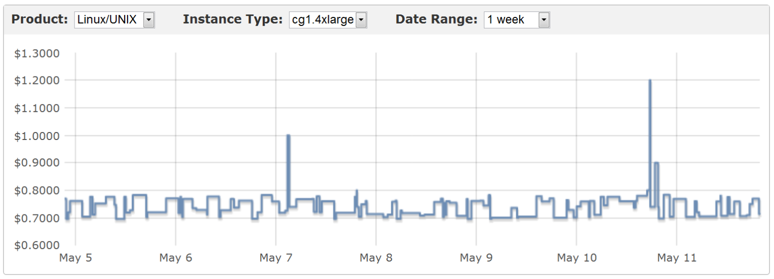
Product: Linux/UNIX Instance Type: cg1.4xlarge Date Range: 1 week

Figure 1: Price History of Auctioned Cloud Resources (Amazon.com: 5/5-11/2011).

- Compiler-based. This approach was designed to avoid the communication “buffering overhead” by generating pseudo dynamic behavior of data matching using a code generator [14].
- Tuple space server-based. This approach has the advantage of ease of programming and performance tuning but carries the extra communication overheads without exploiting their full potentials ([15], [16] and [17]).
- High-level programming language-based. Cilk [18] is a higher level parallel programming language. The philosophy behind Cilk is that a programmer should concentrate on structuring her or his program to expose parallelism and exploit locality, leaving Cilk’s runtime system with the responsibility of scheduling the computation to run efficiently on a given platform.
- Infrastructure-based. The Hadoop system [1] was designed to tackle data intensive large scale loosely coupled applications. Its applications are called “map-reduce” applications that use “divide-and-conquer” algorithms to complete extremely large jobs quickly and reliably. Its runtime system tracks and recovers failed tasks automatically.

Implicit parallel programming has reduced programming complexity and the potentials to offer cheaper fault tolerance. However,to date, except for the Hadoop system, most implicit parallel programs have yet to demonstrate superior performances.

2.2. Fault Tolerant Computing

Applications that can sustain frequent failures can reduce actual computing costs. Figure 1 shows the history of bidding prices of high performance GPU instances with 4xlarge memory in May 2011 at the Amazon cloud. An out-of-bid failure occurs when an instance is acquired by a higher bidder.

Figure 2 shows the relationship between the application’s uptime and bidding prices. For example, assuming zero CPR overhead, for an application requiring 10 hours of computing time using 100,000 instances, the bidding price of 56 cents/hour/instance would encounter one out-of-bid failure on average, or MTBF 5 hours. Bidding 53 cents/hour/instance would generate frequent out-of-bid failures (MTBF) in less than an hour interval. For this task, the cost difference is about $30,000.

There are also hardware failures. Although component failure in a single modern processor is a small probability event, when massively many processors are networked together, the cumulative effects are formidable. Garth Gibson observed that despite of the development slowdown

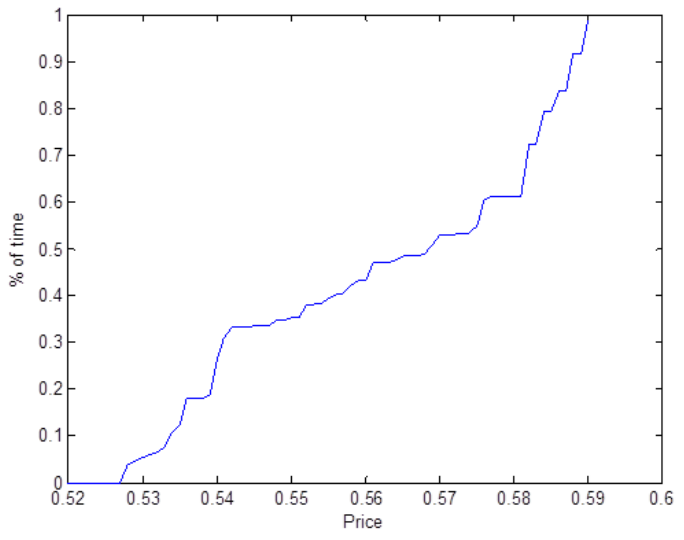

Figure 2: Uptime Probability vs. Bidding Prices.

of faster cycle processors, the number of sockets (cores) per supercomputer would double every year or so. Figure 3 shows three growth models that doubles socket counts every 18, 24 and 30 months [3].

The Petascale Data Storage Institute (PDSI) at Carnegie Mellon University (http://www.pdsi-scidac.org/) has collected and analyzed historical hardware failure data and observed the following [19]: the hardware failures lie in between a rate of 0.1 and 0.25 per year per processor regardless of the configuration of the machine. Since the failure rate of the CPU itself is very small, the failures can be assumed to be tied to sockets rather than processors. Figure 4 shows the projected mean time to interrupt (MTTI) for the three socket growth models in Figure 3.

Currently, the lack of scalability in HPC applications has driven their MTBFs down to 60 minutes [2], known as the “one hour barrier”. This means that for certain large scale applications today it is not possible to have a full hour failure-free run using the current combination of hardware and software. The problem worsens if the application up scales in size.

To preserve the intermediary results, checkpoint-restart (CPR) [20] is necessary. CPR requires periodical savings of application’s intermediate states. When the application crashes (any transient component failure can cause this to happen), we can restart the application from the last checkpoint, thus preserving the energy that had generated the results.

For HPC applications, there are two kinds of CPR: *system level* and *application level*. System level CPR is provided by the parallel programming API (Application Programming Interface) and its processing environment that allows the application program to call for a checkpoint using a single instruction. Recovery is automatic. The Berkeley’s BLCR library [21] is an example of system level CPR. Application level CPR [22] is provided by the programmer who must use his/her understanding of the program to find the suitable time and frequency to save critical data sets. The programmer is also responsible for the coding of application recovery after failure.

In a typical many-core or multi-core parallel system, the host processor must use shared

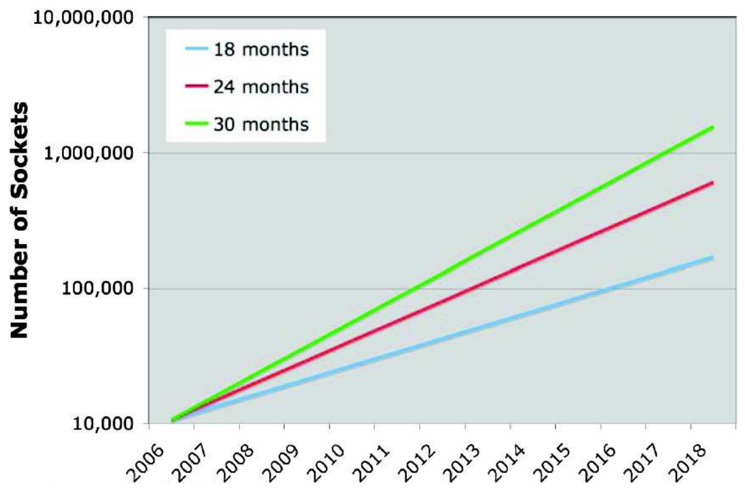

Figure 3: Socket Count Growth Trend in Supercomputers.

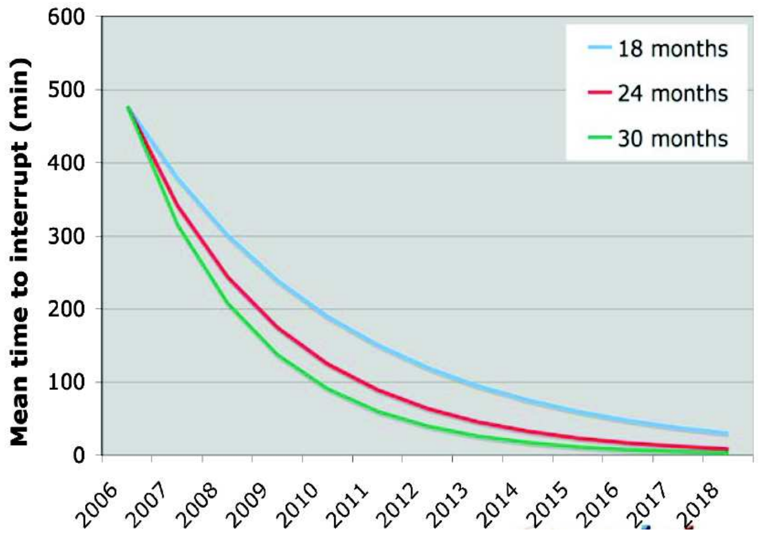

Figure 4: MTTI Trend Prediction by Garth Gibson in 2007.

memory to communicate with multiple sub-processors. A system level checkpoint must save the memory contents accessed by all processors. At the present time, this type of checkpoint is considered a non-trivial challenge ([23], [24] and [25]).

Existing standard HPC frameworks are explicitly parallel, such as the message passing standards [26] and shared memory standard OpenMP [12]. Variants include OpenMPI [27], MPICH [28], MPICH-V [2] and MVAPICH [29]. These systems can support checkpointing the state of individual nodes using the BLCR library and for checkpointing and re-constructing single or multiple level communication states using sophisticated techniques ([22], [2]).

In practical production codes, only application level CPR is widely used. The optimal CPR interval can minimize the overall running time and save the energy consumption by committing the minimal number of checkpoints. Finding and implementing the optimal CPR interval requires non-trivial calculations ([30], [31]). Even with the optimal intervals, time and energy losses are inevitable. For general purpose HPC, fault tolerance at scale is considered very difficult [32].

Since CPR relies on the stable storage for backup data, the pressure on storage is mounting. Even though hardware technologies have improved circuit reliability from historical 0.25% per year per socket to 0.1%, Garth Gibson [19] predicted the following in 2007:

- Top500 supercomputers will double peak FLOPS annually.
- Cycle time will be flat while number of cores on chip grow on Moore's Law.
- Mean Time To Interrupt (MTTI) failure rate will be up 25% to 50% per year.

The actual failure rates had met and exceeded his prediction after the introduction of general purpose GPU (GPGPU) for HPC applications.

It is commonly recognized that none of the above mentioned fault tolerance methods would be capable to contain the rapid growth of failure rates.

In contrast, we observe that the low level point-to-point packet switching network can tolerate multiple component failures and can gain performance and reliability at the same time if including more switches and routers. A curious question is: Why can't we deliver HPC application's sustainability using the same techniques?

3. Motivations

The structural advantages of packet switching networks are the primary motivation of the reported methodology. Unlike HPC application networks, a packet switching network generally allows unlimited scalability in performance and reliability as the network component count increases. Since all HPC applications must involve communications, if we can re-formulate the HPC computing failure problem as a communication failure problem, the sustainable HPC application challenge may have a practical solution. We paid special attention to the timeout treatments in HPC applications. Due to the lack of easy alternatives, an HPC application transmission timeout is treated as a fatal error. Therefore, the entire application will halt on every transient component failure. Since the number of transient failures increases proportionally as the system grows its component count, HPC application's MTBF must decrease accordingly.

To decouple the transient component errors from a HPC application, the application programs should be decoupled to allow statistical multiplexing on the exchanged data. Unlike the packet switching networks, the sustainable HPC application network requires statistical multiplexing of redundant computing and communication components. Like in the packet switched

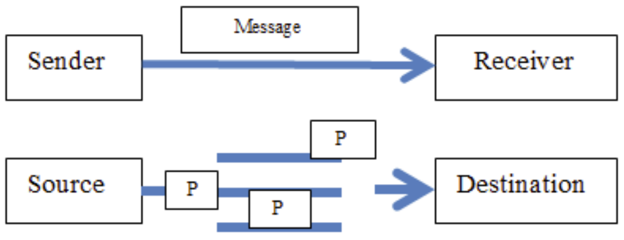

Figure 5: Messaging network and packet-switching data network.

network, the HPC re-transmission logic must also be coupled with idempotent processing plus a statistically multiplexed infrastructure. These constitute the core of the proposed tuple switching network. Historically, the counter-intuitive store-and-forward packet switching network has helped to deliver the most sustainable man-made architecture in human history: the Internet.

The key concepts in a successful sustainable applications network seem to include three elements:

- Find a service dependent unit of transmission.
- Develop an end-to-end protocol with re-transmission and idempotent processing based on the unit of transmission.
- Develop a statistical multiplexed infrastructure based on the unit of transmission.

Without statistical multiplexing, the actual state of a communication task is theoretically not confirmable ([33], [34] and [20]). With statistical multiplexing, the probability of success increases proportionally as the number of redundant computing and communication paths. The re-transmission protocol includes implicit transient storages for the communicating data. Therefore, increasing the networking component counts improves the application network's performance and reliability at the same time. Although the packet-switching overhead is significant compared to direct circuit-switching protocols, the economy of scale in low cost fault tolerance and unlimited scalability of packet switching concept have been proven effective and widely accepted in practice.

One wonders why the low level data network benefits cannot be better exploited by higher level applications without structural changes at the application level. The answer is that they operate on different units of transmissions (Figure 5). Since the mutual information between the HPC application data and the low level packets is near zero [35], although the lower layer packets are automatically multiplexed (if packet switched), the complete application objects are only transmitted once. Thus they are semantically identical to datagrams in low level UDP data networks. By the end-to-end principle [9], not only a statistically multiplexed infrastructure is necessary, but also the application programming interface (API) must include the semantics of statistical multiplexing for the application to be sustainable.

In other words, we are interested in an "infrastructure-based" implicit parallel programming paradigm, we call "tuple switching network". The proposed system will rely on the application generated data flows to dispatch parallel programs automatically. The semantic statistical multiplexing infrastructure ensures zero single point failure and application sustainability at the same time.

Ultimately, since the application semantic network decides the final results of the HPC application, we argue that without semantic statistical multiplexing, it is not possible to eliminate all single-point-failures in the applications network. And it is also not possible to gain performance and reliability at the same time as we add resources.

In comparison to the Hadoop system [1], the proposed tuple switching network is a generalization that is capable of extreme scale generic HPC applications and extreme scale data intensive applications.

4. Tuple-Driven Parallel Processing

In a tuple-driven parallel processing environment, application dependent data tuples are stored and forwarded by a tuple space infrastructure. Automatic worker failure protection can be provided by the tuple space implementation to allow a retrieved tuple to assume "invisible" status until its processing is completed. If the corresponding result does not arrive in time, its "invisible" status can be reversed to "visible", allowing other available computing nodes to compete for the unfinished task [30]. In view of the tuple switching network, this mechanism satisfies statistical multiplexing requirement analogous to the automatic re-transmission of TCP packets.

The tuple-driven parallel processing environment has the identical semantics of a data driven parallel processing model. This saves us from providing the feasibility arguments.

Since most high power many-core and multicore processors are exclusively used for parallel workers, this allows us to use tuple parallel programming to meet the system level many-core and multicore processor checkpoint challenges.

Unlike explicit parallel programming methods, tuple-driven parallel programming relies on communicating tuples to automate task activation (also called "firing" in literature [6]). This enables automatic formation of SIMD, MIMD and pipeline clusters at runtime [6]. The net-benefit of this feature is automatic "hiding" of communication latencies (or "work stealing" [18]). For extreme scale HPC applications, these qualities help in reducing parallel programming complexities.

In addition to "stateless" workers, a practical HPC application must also contain "stateful" masters (programs responsible for delivering the semantically identical results as sequentially processing the same data). The masters must still be checkpointed to preserve the intermediate results. It is not immediately clear if the overall computing time with multiple master checkpoints (although less frequent and potentially smaller) would still deliver sustainable savings given the potentially slower tuple switching network.

A tuple-driven programming environment forces the programmers to focus on data partitions to expose parallelism in proper granularity. The data partitioning strategy determines the ultimate deliverable performance. There is no need to manually manipulate concurrent tasks.

The tuple-based API does not have a fixed process-data binding. Therefore, it is possible to statistically multiplex the application data in between. Specifically, if the tuple-based API contains re-transmission and idempotent processing for application tuples, supported by a robust tuple space implementation, all-around scalability must be attainable.

The tuple-driven applications require the processing environment to support a tuple space abstraction for matching computing tasks at runtime. This is the source of additional communication overhead which almost doubles the cost for every direct inter-processor communication request. However, these overheads have enabled new important possibilities for extreme scale applications:

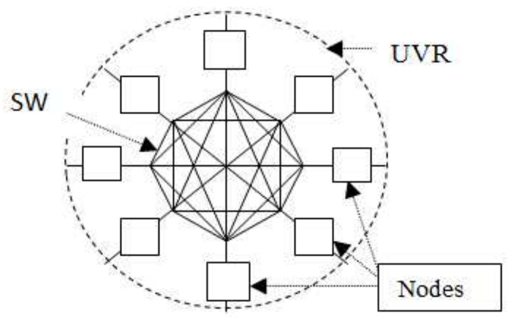

Figure 6: Tuple Switching Network.

1. The possibility to deploy multiple interconnection networks in parallel, thus relieving the pressure on the single interconnection network to allow more computing nodes in parallel and to support diverse communication patterns.
2. The possibility to offer automatic worker fault tolerance, thus reversing the negative effect on application MTBF and delivering sustainable performances.

The tuple-based API also makes it easier to seamlessly incorporate multiple heterogeneous processor types, such as single-core, many-core and multicore CPUs, DSPs, and GPGPUs. It can also include legacy HPC applications. This is the basis for the proposed two-tier tuple switching system.

Like the Internet, a tuple switching-based two-tier network can potentially deliver scalable performance, scalable availability and energy efficiency at the same time.

The following sections reports the design and implementation of tuple switching network and findings based on an experimental tuple-based statistical multiplexing project named Synergy [36].

5. Tuple Switching Network and Stateless Parallel Processing

5.1. Architecture

Tuple switching network requires a data-driven parallel processing paradigm. Unlike existing command-driven paradigms, data parallel programs must assume zero global state. In this sense, data-parallel programs can be considered stateless parallel programs (SPP) [37]. Figure 2 shows the conceptual architecture of the proposed tuple switching network.

In Figure 2, SW represents a collection of redundant networks. UVR stands for Unidirectional Virtual Ring – a fault resistant (self-healing) virtual communication channel that links all nodes for an application. Each node is a standalone processor of some particular type. Each node has multiple network interfaces, local memory, disk and single or multiple processing units; it can also host multi-core or many-core processors.

The global tuple space is implemented as follows:

1. Data requests are matched locally. Unmet requests travel through UVR for global match.
2. All nodes participate in data matching in parallel.
3. All networks participate in direct exchange of matched data.

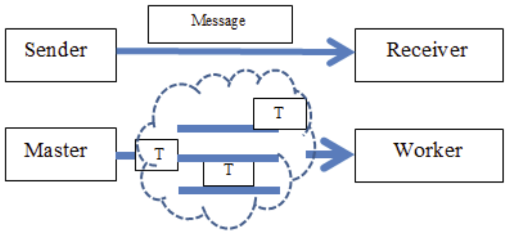

Figure 7: Static and statistical semantic networking.

These functions are implemented in a single daemon that runs on each node. Like typical peer-to-peer systems, these daemons communicate with each other to form a single consistent HPC machine image using all runtime available resources. Each daemon implements a local tuple space for programs running on the node. Global data matching requests travel by UVR. The actual global data transfers are done in parallel via the multiple redundant physical networks. Running applications with optimized grain sizes should have only a few tuples residing on each node. At the application level, all nodes participate in a statistically multiplexed global tuple space. Programs use the tuple space API to communicate with a local daemon which in turn communicates with other daemons to complete data acquisition in parallel if the request cannot be met locally. Each application exploits multiple redundant networks automatically to counter-balance the inherent speed disparity between computing and communication components. There is no single point failure for such HPC applications. Using a binary broadcast protocol, each UVR can scale to include millions of nodes with no more than $O(lgP)$ data matching complexity.

Failure containment for multiple multicore and many-core processors is now feasible by leveraging the automatic "worker fault tolerance" without involving low level memory CPR. Unlike traditional supercomputing environments, the statistical multiplexed semantic network promises enhancing application performance and reliability at the same time by adding redundant computing nodes and networks.

Figure 3 makes a conceptual comparison between the explicit and implicit application networks, where *T* stands for "tuple" which is the unit of transmission of the HPC semantic network.

In Figure 3, the explicit parallel programming API produces a static application network. Each transmission is semantically identical to a UDP datagram in lower level networks: it only gets sent once. The implicit parallel programming API allows the tuples in the semantic network to form TCP-like virtual circuits with automatic re-transmission and idempotent processing by the tuple switching infrastructure (tuple space).

5.2. Application Development

A data-driven HPC application will use only data tuples for inter-program communication. The Tuple Space abstraction [8] is a natural representation.

The tuple space API contains three data manipulation primitives [37]:

1. Put(TupleName, buffer): This call inserts the contents of "buffer" with TupleName into the space.
2. Get(&NameBuffer, &buffer): This call retrieves and destroys a tuple with a matching name in NameBuffer.
3. Read(&NameBuffer,&buffer): This call only retrieves a tuple with a matching name in NameBuffer.

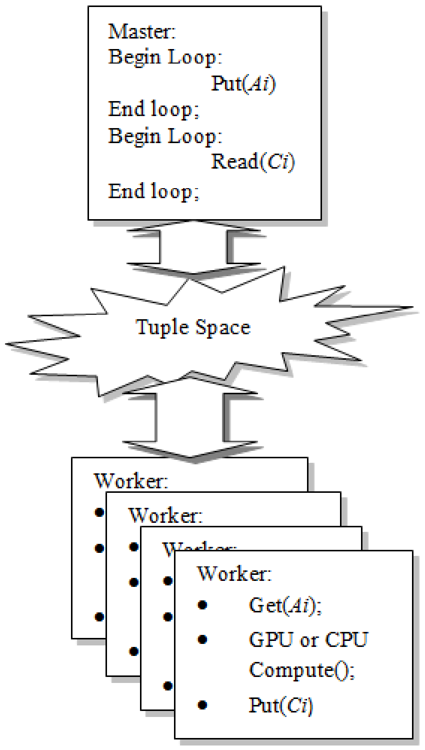

Figure 8: Logical View of SPP Programming.

The “&” sign represents “access by reference” meaning that the variable NameBuffer’s contents can be altered to hold the value of a matching tuple name at runtime.

Since different processor types and processing environments require different coding (MPI, OpenMP, CUDA, etc), each worker can contain multiple implementations for the same kernel in order to adapt itself to the available resources at runtime.

Figure 4 illustrates the programmer’s view of tuple space parallel processing. Each application will be decomposed into multiple masters with each responsible for a computing intensive kernel in the application. Each master program is matched with a single worker program. Each worker program will run automatically on multiple available heterogeneous processors.

The master program uses the “Put” command to send unprocessed work assignments to the tuple space. It uses the “Get” command extracts the results.

The worker program repeats the “Get”, “Compute” and “Put” sequence for as long as there are assignment tuples. Since the worker codes are programmed to automatically adapt to different processors, it will run on all nodes accessible at runtime. The application terminates when there are no more assignment tuples.

The correct user data handling relies on statistical multiplexing of redundant computing and networking resources. Without an infrastructure supported unit of transmission, it is not possible to implement statistical multiplexing correctly using explicit parallel programming APIs or compiler-based implicit parallel programming methods, such as the approaches taken by the

Linda project [8] and others [18].

5.3. Sustainability Analysis

In this section, we assess the application's sustainability by its expected time savings using "worker fault tolerance", as promised by the statistical multiplexed tuple switching network.

To do this, we build two models (based on [30]) for a typical HPC application with checkpoints. The first one is for explicit parallel programming systems where any component failure would cause the entire application to halt. The second is for implicit parallel programming, where only master failure or 100% worker failure would halt the application. We then compare the expected processing times using the respective optimal checkpoint intervals. It is worth mentioning that unlike [31] where the optimal checkpoint interval model was based on a system exhibiting Poisson single component failures, the following models assume multiple Poisson component failures.

According to [30], we define the expected computing time with failure, as follows:

- t_0: Interval of application-wide checkpoint.
- α: Average number of failures within a unit of time which follows Poisson distribution.
- K_0: Time needed to create a checkpoint.
- K_1: Time needed to read and recover a checkpoint.
- T: Time needed to run the application without checkpoints.

Further, we define:

- α_1: Average number of failures of critical (non-worker) element failure in a time unit which follows Poisson distribution.
- α_2:Average number of failures of non-critical (worker) element failure in a time unit which follows Poisson distribution.

Thus, $\alpha = \alpha_1 + \alpha_2$.

Assuming failure occurs only once per checkpoint interval and all failures are independent, the expected running time E per checkpoint interval with any processing element failure is

$$E = (1 - \alpha t_0)(K_0 + t_0) + \alpha t_0(K_0 + t_0 + K_1 + \frac{t_0}{2})$$

.

The expected running time per checkpoint interval with worker failure tolerance will be:

$$E' = (1 - \alpha t_0)(K_0 + t_0) + \alpha_1 t_0(K_0 + t_0 + K_1 + \frac{t_0}{2}) + \alpha_2 t_0(K_0 + t_0 + X)$$

.

where X = recovery time for worker time losses per CPR. We can then compute the differences $E' - E$, as follows:

$$\begin{aligned}E - E' =&(\alpha - \alpha_1)t_0(K_0 + t_0 + K_1 + \frac{t_0}{2})\\ &- \alpha_2 t_0(K_0 + t_0 + X)\\ =&\alpha_2 t_0(K_0 + t_0 + K_1 + \frac{t_0}{2} - K_0 - t_0 - X)\\ =&\alpha_2 t_0(K_1 + \frac{t_0}{2} - X)\end{aligned}$$

Since the number of workers is typically very large, the savings are substantial. The total expected application running time E_T without worker fault tolerance is:

$$E_T = \frac{T}{t_0}(K_0 + t_0 + \alpha(t_0 K_1 + \frac{t_0^2}{2}))$$

We can now compute the optimal checkpoint interval:

$$\frac{dE_T}{t_0} = T(-\frac{K_0}{t_0^2} + \frac{\alpha}{2})$$
$$t_0 = \sqrt{(\frac{2K_0}{\alpha})}$$

The total application running time E_T with worker fault tolerance is:

$$E_T = T(1 + \frac{K_0}{t'_0} + \alpha K_1 + \frac{\alpha t'_0}{2} - \alpha_2 K_1 - \frac{\alpha_2 t'_0}{2} + \alpha_2 X)$$

The optimal checkpoint interval with worker fault tolerance is:

$$\frac{dE_T}{t'_0} = T(-\frac{K_0}{t_0'^2} + \frac{\alpha - \alpha_2}{2})$$
$$t'_0 = \sqrt{(\frac{2K_0}{\alpha - \alpha_2})}$$

For example, if we set the checkpoint interval $t_0 = 60$ minutes, the checkpoint creation and recovery time $K_0 = K_1 = 10$ minutes, and the average worker failure time loss $X = 30$ sec or 0.5 minute, the expected savings per checkpoint under any single worker failure is about 39.5 minutes (or greater than 50% savings).

$$\begin{aligned}E - E' &= \alpha_2 t_0(K_1 + \frac{t_0}{2} - X)\\ &= (10 + 30 - 0.5)\\ &= 39.5,\end{aligned}$$

because $\alpha_2 t_0 = 1$ (single worker failure).

On the other hand, if the MTBF is 3 hours in a system of 1024 processors, this gives $\alpha t_0 = 180\alpha = 1$ or $\alpha = 1/180$. Thus, $\alpha_1 = 1/(180 * P) = 1/184,320$. The optimal checkpoint interval for a system with a single master and 1024 workers would be:

$$t'_0 = \sqrt{(\frac{2K_0}{\alpha - \alpha_2})} = \sqrt{2 \times 10 \times 184320} = 1,920.$$

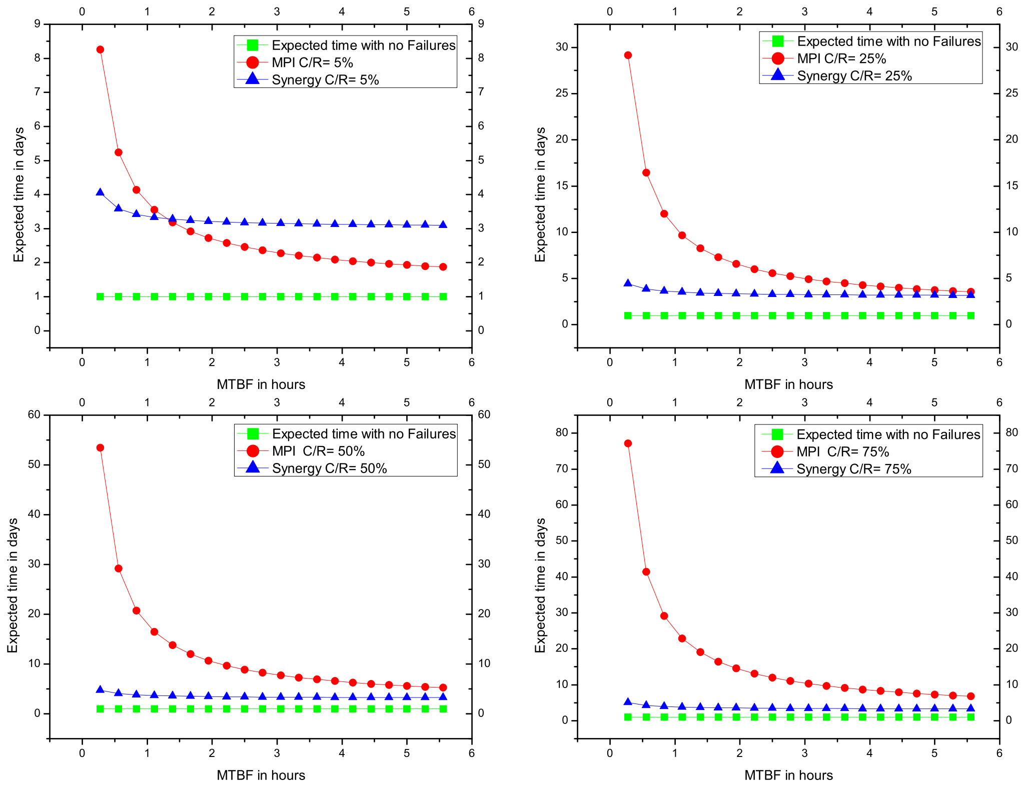

Figure 9: Sustainability analysis: Expected Elapsed Time vs. MTBF (P = 1024, Tuple Space slowdown factor = 3).

This means that for this HPC application using 1024 nodes, it is not necessary to checkpoint the master unless the application running time T is greater than 30 hours.

In terms of energy savings, for an application that needs 30 hours computing time, the total energy savings would be about 5 Megawatt hours (1024 processors with 187 watts per many-core GPU processor).

Assuming the statistical multiplexing (tuple space) slow down factor = 3, Figure 5 shows the expected time savings (T) versus the application MTBFs as the CPR overhead (Checkpoint Time/Running Time) varies from 5%, 15%, 25% to 35% for a processor of 1024 nodes. In Figure 5, higher CPR overheads or lower MTBFs deliver bigger performance advantages.

These figures also suggest the practical break point between using single tier mono-scale simulation and two-tier, possibly multi-scale simulation. The tuple switching network should be introduced in the second-tier to contain the risks of lower tier failures. The lower tier is a collection of concurrently running legacy HPC production codes solving multiple subproblems. Since multicore and many-core processors are almost exclusively used for workers, the proposed paradigm offers system level multiple multi-core/many-core processor fault tolerance without involving checkpointing low level shared memories.

6. Computational Experiment

6.1. Experiment Setup

Application. For simplicity, we use matrix multiplication to simulate the compute intensive core of a large scale time marching simulation application. Given two $N \times N$ matrices A_0 and B, the experimental system computes k matrix products as follows: $(0 < i \leq k)$:

$$\begin{aligned} C &= A_k \times B, \\ A_i &= A_{i-1} \times B \end{aligned}$$

C is the final solution. We then created one MPI and one Synergy implementation for the same application. Both implementations include a master and a worker.

For the MPI program, granularity is set to $\frac{N}{P}$ (P=number of processors). The Synergy program granularity tuning range is the top loop of the three nested iterations (see Appendix A).

Objectives. We would like to compare the actual running times of MPI and Synergy implementations with and without failures. We record the following information:

1. Elapsed time without checkpoints.
2. Elapsed time with checkpoints without failure.
3. Elapsed time with checkpoints and recoveries with injected failures.

Processing Environment. We used the Lincoln cluster by NCSA, hosted at Teragrid (www.teragrid.org), for the reported experiments. The Lincoln cluster consists of 192 compute nodes (Dell PowerEdge 1950 dual-socket nodes with quad-core Intel Harpertown 2.33GHz processors and 16GB of memory) and 96 NVIDIA (many-core) Tesla S1070 accelerator units. Our application allows 20 Tesla units. Each unit has 8 CPUs with 2.33 GHZ each, 16GB memory total, and 4 Tesla S1070 cards. Each unit provides 345.6 gigaflops of double-precision performance. The file system is Lustre with 400 TB disk storage shared with another cluster (Abe).

Development Software. All experiments run in Red Hat Enterprise Linux 4. The GPU codes use CUBLAS (CUDA 2.2) and Intel C++ compiler 10.0 for Linux.

The parallel processing environments include:

- OpenMPI (http://www.open-mpi.org/)
- Synergy v3.0 (http://spartan.cis.temple.edu/synergy)

Failure Injection Method. The minimal number of processors we would like to validate our calculations with is 1024. We distribute the failures to the optimal number of GPU units for each environment.

We use a Poisson random number generator, as in [38], to perform the injection of failures. The failure injection algorithm accepts variable MTBFs.

6.2. Computation Experiment and Results

In practice, the checkpoint creation time K_0 is different for MPI and for Synergy. For MPI, the checkpoint must include the global state of all involved nodes. For Synergy, the master checkpoint only needs to include local states. Multiple masters will checkpoint in sync and in parallel using a distributed synchronized termination algorithm [39].

The recovery time K_1 is also different. For MPI, the recovery time covers reading of a globally saved state. For Synergy, the recovery time includes launching multiple masters reading

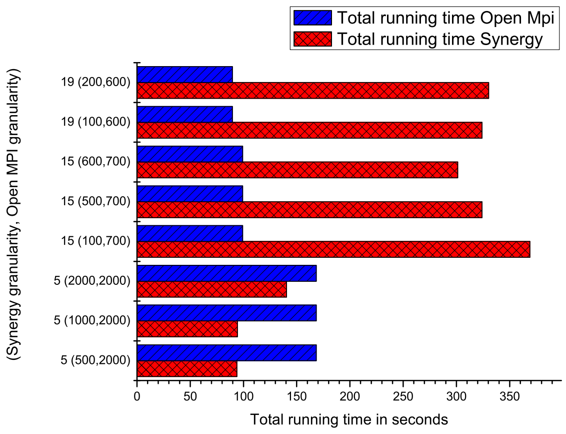

Figure 10: Performance without checkpoint and failure (N = 10,000; Rounds = 10).

the multiple saved states in parallel. For simplicity, we consider the differences negligible and a single master is necessary.

The matrix multiplication kernel is programmed using CUDA linear library CUBLAS. It is included in the CUDA 2.2 toolkit. The CUDA kernel is “wrapped” by the Synergy calls as depicted in Figure 4.

GPU programming is very sensitive to the change in the granularity due to loading overheads. Fine tuning granularity produced counter-intuitive results, shown in Figure 5 ($P = 5$), where MPI granularity is fixed $\frac{N}{P}$.

Figure 6 also shows that the best time to solution for the MPI implementation is 19 GPU workers (granularity = 600). The best time to solution for the Synergy implementation is 5 GPU workers (granularity = 1000).

Both the MPI and the Synergy workers can be programmed to adapt to either CPU or GPU processors at runtime based on the availability of a free device. In this experiment we used workers that only finds GPU devices, locking it and using it to do the matrix computation.

As mentioned earlier, system level CPR for multiple GPUs is an unsolved challenge for MPI codes. We had to use application level checkpointing. The same CPR code is used for Synergy master, where worker fault tolerance is provided by automatic “shadow tuple” recovery [37].

The computation results were recorded runs with the following parameters:

- $N = 10,000$.
- $P = 5$ or $P = 19$ (the best time to solution for Synergy and MPI respectively).
- $K = 400$ (rounds).
- K_0 is measured 10 seconds.
- t_0 and t_0'(optimal CPR interval) are calculated automatically for each scenario.

The failure injection algorithm is tunable for different MTBF values. We then distributed the projected failures (based on $P = 1024$) across all processing nodes statistically.

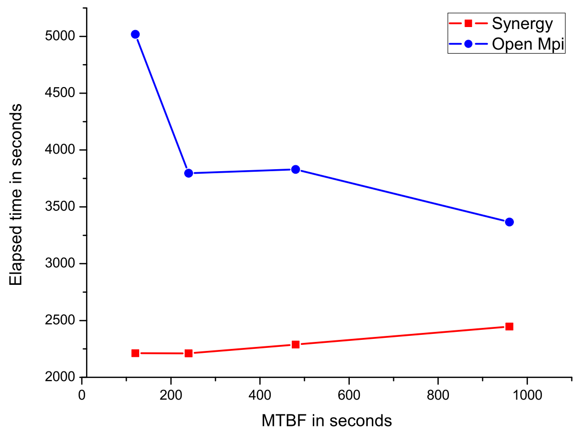

Figure 11: Running time with failures. N = 10,000; Rounds = 400; P = 5(Synergy); P = 19(OpenMPI).

Figure 6 shows the performance differences between OpenMPI and Synergy without checkpoints. For Synergy, the shadow-tuple capability is always enabled. It shows that for small number of GPU units (5), Synergy out performs MPI due to granularity optimization. MPI beats Synergy performance at larger P values. For the same amount of work, MPI program needed more GPU resources.

As mentioned earlier, our application level checkpoint simply writes the matrices to the stable storage synchronously (to avoid restart errors). Otherwise, we would lose the latest checkpoint due to the disk caching. All checkpoints are executed at the dynamically calculated optimal intervals according to the number of processors used and the discussions in Section 5.

Failure were injected by a “killer” program. The killer program runs at the end of each MTBF cycle (Figure 6). It then kills a random running process.

For the MPI run, each random kill is “all or nothing”. This means that if the “killer” needs to terminate any process, the master must reload the last checkpoint file and lose all the rounds computed since. Since the reloading is mandatory, this setup produces statistically equivalent results as for $P = 1024$.

In the case of the Synergy run, the master CPR process is similar to MPI except that the failures are statistically distributed as master or worker failures. A master failure will follow the same process as MPI, but a worker failure does not stop the running. Once a worker is killed the work tuple that was assigned to this worker reappears in the tuple space after a short delay. A new computing node will pick up the load. This allows the overall computation to continue with limited time loss (variable X as discussed in Section 5).

Figure 7 shows the computational results for OpenMPI and Synergy with injected failures. We accelerated MTBFs to explore the potential impacts of higher failure rates. The results are consistent with the sustainability model predictions.

7. Discussions

This paper reports a tuple switching network for sustainable generic HPC applications at scale. We proposed an Internet-like two-tier tuple switching network in order to mitigate the increasing risks of massive component failures and to gain performance scalability at the same time.

We argue that explicit parallel programming is not appropriate for extreme scale applications for its inability to handle uncertainties. The current standard benchmarks are the best practices of explicit parallel programming. These results, however, can be leveraged for extreme scale application if integrated with the proposed tuple switching network at the higher level.

The tuple switch network also brings the possibility of parallel performance optimization. Finding the optimal processing granularity is a tricky problem for explicit parallel systems like MPI and OpenMP, since every change in granularity requires re-coding. In implicit or dataflow environments, parallel programs and their data are decoupled in time and in space. This gives the application network layer a chance to handle uncertainties. Granularity change does not require re-coding if the tuning range is within the same loop index. A linear coarse-to-fine grain optimal granularity finding process (outermost loop $\Rightarrow$ innermost loop) can identify the optimal partition depth (tuning range). Since the ratio between aggregate computing and communication directly affects the optimal partition depth, instrumentation and experimentation are also required. For the matrix multiplication example above, we found the optimal granularity for the NCSA Teragrid is still the top loop (see Appendix A). Fine turning the grouping factor within the top loop produced better performance than its MPI counterpart when P = 5. Note that the Synergy performance includes the checkpoint (shadow tuple) overheads for all worker nodes.

In anticipation for future technology advances, ideally, the "best" parallel programs should be able to change processing granularity without changing code at all, since hand-tuned parallel codes will quickly lose their performance advantage as soon as the processing environment changes. The need for tunable and sustainable HPC code also becomes more evident if we want to leverage auction-based cloud resources, since applications that can tolerate higher out-of-bid failures will cost far less than those that do not. To minimize manual work, we have also studied an automatic data parallel code generation method using a simple parallel markup language (PML) [40] for cross level performance optimization.

If we consider each packet buffer as a CPR point for data service, we can then envision extreme scale applications using the same strategy by eliminating "state spreading" and by statistically multiplexing semantic communicating objects. The standard benchmarks are still relevant for they have set the baseline of the best performing "fat nodes". Using the proposed two-tier framework, we can deliver extreme scale performances by statistically multiplexing multiple "fat nodes" and multiple interconnection networks that link them.

The advent of auction-based cloud computing further stimulates the need for sustainable HPC applications, since better costs are delivered by lower bidding prices (higher out-of-bid failures) (Figure1). With tuple switching framework, it is possible to define the "optimal" cost for a given HPC application based on a given budget, time or both.

Sustainable applications are also energy efficient. The savings come from drastically reduced CPR overheads and improved resource efficiency. The proposed tuple switching network touches power efficiency, programming complexity and failure tolerance aspects of extreme scale HPC applications. As hybrid HPC systems have demonstrated more efficiency and cost effectiveness than homogeneous systems [3], the proposed framework fits well into the technology evolution trajectory.

For some applications, such CFD (computational fluid dynamics) codes, to gain application sustainability, the mutually dependent stateful workers can be converted to stateless workers by translating the "neighboring" dependencies via automated tuple matching. The application will lose some communication efficiency (as addressed in the paper by the slowdown factor), but it will gain overall sustainability.

There are also more potentials for productivity gains. For example, since explicit parallel programming is easier, it may not be necessary to manually write the tuple switching "wrappers" for legacy applications for building the extreme scale application. Automated tools have been experimented using Parallel Markup Language (PML)[40]. More efforts would be needed to study how to compose multi-scale codes using the proposed tuple space parallel programming environment [41]. This also applies to the GPGPU applications where skilled programmers would focus on producing the optimized GPU kernels while the wrapper would be automatically generated.

HPC application batch scheduling would also be greatly simplified since the tuple-driven applications will exploit and optimize all available resources automatically. Similar to map-reduce applications, there is no need to compute sophisticated resource scheduling policies. Non-stop HPC application would become a reality where components can be taken offline for repair without shutting down the running applications. Resource efficiency would further improve since we can now afford the optimal processing granularity and optimal checkpoint intervals.

Future studies would also include research on diverse applications with different communication patterns. Automatic matching of interconnection network topologies with runtime communication patterns would also be possible.

With commercial cloud computing a reality, minimal cost HPC computing is possible using auction-based resources. The tuple switching network can comfortably handle the out-of-bid failures to deliver more cost-effective solutions under practical constraints. Optimization models are needed to help users to calculate the best strategy to maximize the yield of a given budget or time. Since the tuple switching network is very similar to the packet switching network, with hardware advances in high performance circuits [42], it also is possible to develop hardware support for application specific networks.

8. Conclusions

The fundamental result of the reported tuple switching network is the use of statistical multiplexing of application data to solve the seemingly "impossible" scalability challenge of extreme scale HPC applications. We have shown that the powers of statistic multiplexing can be leveraged to tame the nagging HPC sustainability challenges that have troubled us for a long time.

Without statistical multiplexing both computing and communication components, it is not possible to eliminate all single-point-failures in the applications networks. And it is also not possible to gain performance and reliability at the same time as we add resources.

The proposed tuple switching network is a generalized infrastructure-based implicit parallel processing paradigm. In comparison to the Hadoop system, the tuple switching network is more general in that it is capable of extreme scale HPC applications and extreme scale data intensive applications (with much lower overheads).

Our computational experiment demonstrated the practical feasibility that confirms with the greater implications described in the sustainability model.

Architecture-wise, the proposed tuple switching network has merely removed the structural impediments for sustainable HPC applications. Research of faster networks and processors are still needed to shorten communication and processing latencies. Like the packet switching network, the tuple switching network can only up scale bandwidth and reduce latency under overloading conditions. The architecture, however, is poised to subsume future advances in computing and communication devices.

Tuple switching network differs from the generic “messaging-switching network” [43] in the semantics of unit of communication. Application semantic multiplexing also requires fundamentally different user data handling than traditional server-based generic tuple space implementations ([16] and [17]). Like the packet switching protocol that has delivered scalable stochastic and reliable deterministic data services, the same can be expected for tuple-based applications using tuple switching network. Optimization of semantic network multiplexing is also possible by cross layer research since most HPC semantic networks are implemented using the TCP/IP protocol. Optimization can help to eliminate unnecessary redundancies.

Tuple switching network is a solution for a type of network applications that are suitable for using data tuples. These include HPC applications and many business and E-commerce applications ([16] and [17]). The general methodology can also be applied to solve sustainability problems for other non-trivial network application types, such as online transaction processing systems, extreme scale storage networks and extreme scale service oriented systems [5]. In theory, if an architecture can be defined to apply statistical multiplexing to the correctly formulated application unit of transmission, the proposed methodology can drastically reduce communication-induced application-level uncertainties – a desirable feature for all large scale mission critical applications. Like Johann Sebastian Bach once said, “.. (if) the keys are pressed at the right times, the instrument will play itself.” Since most applications naturally gravitate towards mission critical status, the concept of statistical multiplexing of semantic network can be important for all future robust extreme scale computing/communication systems.

Acknowledgments

The authors thank Dr. Axel Kohlmayer for the introduction to Teragrid resources. This research is supported in part by a grant from the National Science Foundation CNS 0958854.

References

[1] Apache, Hadoop tutorial, 2011.
[2] F. Cappello, Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities, International Journal of High Performance Computing Applications 23 (2009) 212–226.
[3] J. Layton, Petaflops for the common man - 5 challenges, 2011. [Online] http://en.community.dell.com/techcenter/high-performance-computing/w/wiki/2279.aspx.
[4] HPC resilience consortium, 2010. [Online] http://resilience.latech.edu.
[5] J. Y. Shi, Chapter 19: Fundamentals of cloud application architectures, Cloud computing: methodology, system, and applications, CRC, Taylor & Francis Group.
[6] J. B. Dennis, Data flow supercomputers, Computer 13 (1980) 48–56.
[7] Arvind, Decomposing a program for multiple processor system, in: Proceedings of the 1980 International Conference on Parallel Processing, pp. 7–14.
[8] N. Carriero, D. Gelernter, How to Write Parallel Programs - A First Course, The MIT Press, Cambridge, MA, 1990.
[9] J. Saltzer, D. Reed, D. Clark, End-to-end arguments in system design, Second International Conference on Distributed Computing Systems (1981) 509–512.
[10] P. Baran, On Distributed Communications, RM-3420, Technical Report, http://www.rand.org/about/history/baran.list.html, 1964.
[11] M. P. I. FORUM, MPI: A message-passing interface standard, 1994.
[12] R. Chandra, et al., Parallel Programming in OpenMP, Morgan Kauffman, 2001.
[13] J. Gurd, I. Watson, A multilayered dataflow computer architecture, in: Proceedings International Conference on Parallel Processing.
[14] S. Ajuja, N. Carriero, D. Gelernter, Linda and friends, Computer 19 (1986) 26–34.
[15] J. Y. Shi, Synergy v3.0 Manual, 2010.
[16] Introduction to xap 8.0 - gigaspaces, 2011. [Online] http://www.gigaspaces.com/wiki/display/XAP8/8.0+Documentation+Home.

[17] Introduction to javaspace technology, 2010. [Online] http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/.
[18] M. Frigo, P. Halpern, C. E. Leiserson, S. Lewin-Berlin, Reducers and other cilk++ hyperobjects, ACM Symposium on Parallelism in Algorithms and Architectures (2009).
[19] G. Gibson, Reflections on failure in post-terascale parallel computing, in: Proceedings of 2007 International Conference on Parallel Processing.
[20] A. S. Tanenbaum, M. V. Steen, Distributed Systems: Principles and Paradigms, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.
[21] P. H. Hargrove, J. C. Duell, Berkeley lab checkpoint/restart (BLCR) for linux clusters, Journal of Physics: Conference Series 46 (2006) 494–503.
[22] M. Shultz, G. B. R. Fenandes, D. M. K. Pingali, P. Stodghill, Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs, in: Proceedings of Supercomputing 2004 Conference, Pittsburgh, PA.
[23] K. K. H. Takizawa, K. Sato, H. Kobayashi, Checuda: A checkpoint/restart tool for cuda applications, 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies (2009) 408–413.
[24] S. Laosooksathit, C. Leangsuksan, A. Dhungana, C. Chandler, K. Chanchio, A. Farbin, Lightweight checkpoint mechanism and modeling in GPGPU, in: Proceedings of the hpcvirt2010 conference.
[25] CUDA-compute unified device architecture, 2010. [Online] http://en.wikipedia.org/wiki/CUDA.
[26] H. Rolf, High-performance computing and networking, in: W. Gentzsch, U. Harms (Eds.), The MPI standard for message passing, volume 797 of *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, 1994, pp. 247–252.
[27] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, concept, and design of a next generation MPI implementation, in: D. Kranzlmller, P. Kacsuk, J. J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 3241 of *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, 2004, pp. 353–377.
[28] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, A. Selikhov, MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes, in: Supercomputing, ACM/IEEE 2002 Conference, p. 29.
[29] M. Koop, T. Jones, D. Panda, Mvapich-aptus: Scalable high-performance multi-transport MPI over infiniband, in: IPDPS 2008 International Symposium on Parallel and Distributed Processing, pp. 1–12.
[30] W. Gropp, E. Lusk, Fault tolerance in message passing interface programs, Int. J. High Perform. Comput. Appl. 18 (2004) 363–372.
[31] J. T. Daly, A higher order estimate of the optimum checkpoint interval for restart dumps, Future Generation Computer Systems 22 (2006) 303–312.
[32] E. N. Elnozahy, J. S. Plank, Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery, IEEE Trans. Dependable Secur. Comput. 1 (2004) 97–108.
[33] M. Herlihy, N. Shavit, The topological structure of asynchronous computability, J. ACM 46 (1999) 858–923.
[34] A. Fekete, N. Lynch, Y. Mansour, J. Spinelli, The impossibility of implementing reliable communication in the face of crashes, J. ACM 40 (1993) 1087–1107.
[35] T. Cover, J. Thomas, Elements of Information Theory, John Wiley & Sons, Inc., 1991.
[36] Y. Yang, Fault Tolerance Protocol for Multiple Dependent Master Protection in a Stateless Parallel Processing Framework, Ph.D. thesis, Temple University, 2007.
[37] J. Y. Shi, Decoupling as a foundation for large scale parallel processing, in: Proceedings of 2009 High Performance Computing and Communications, Seoul, Korea.
[38] A. M. Law, W. D. Kelton, Simulation Modeling and Analysis, McGraw-Hill Higher Education, 2nd edition, 1997.
[39] B. Szymanski, J. Y. Shi, N. S. Prywes, Synchronized distributed termination, IEEE Trans. Softw. Eng. 11 (1985) 1136–1140.
[40] F. Sun, Automatic Program Parallelization Using Stateless Parallel Processing Architecture, Ph.D. thesis, Temple University, 2004.
[41] Multi-scale modeling and simulation, 2010. [Online] http://www.math.princeton.edu/multiscale/.
[42] High performance FPGA development group, 2010. [Online] http://www.fhpca.org/.
[43] J. F. Kurose, K. W. Ross, Computer Networking: A Top-Down Approach Featuring the Internet, Addison-Wesley, 2000.

Appendix A. Application Source Codes

```
/*----------------------------------------------------------------------------
                       Matrix Multiplication Master
----------------------------------------------------------------------------*/
#include <stdio.h>
#include <synergy.h>
double wall_clock();
double A[N][N],ituple_B[N][N], C[N][N];
char tpname[20];
double *ituple_A; /* [0] = grain size, [1] = values per rows ... */
double *otuple;   /* [0] = grain size, [1] = values per rows ... */

main()
{
    char host[128];
    int i, j, k, received, ix, iy, tplength, status, G, T, R, P, res, tsd, x;
    double t0, t1;
    float F;

    gethostname(host, sizeof(host));
    t0 = wall_clock(); ix = 0;

    // Open Tuple Space Objects
    tsd = cnf_open("problem",0); res = cnf_open("result",0);

    // Retrive command line arguments
    G = cnf_getf(); P = cnf_getP();  // Granularity and P
    if (G > N/2) { // Prevent over chunking
        printf("Chunk size too large (Max: %d)\n",(int) N/2); exit (1); }

    // Create test data
    tplength = (1+N*N)*sizeof(double);
    for (i = 0; i < N; i++) for (j = 0; j < N; j++) {
         ituple_B[i][j] = (double) i * j;
         A[i][j] = (double) i * j;}

    // Insert Matrix B
    sprintf(tpname,"B%d\0",0);
    status = cnf_tsput(tsd, tpname, (double *)ituple_B, tplength);
    // Insert Matrix A in groups
    tplength = (1+ G*N) * sizeof(double);
    if ((ituple_A = (double *)malloc(tplength)) == NULL) exit(1);
    R = N;
    while (R > 0) {
        if (R < G) G = R;
        R = R - G ; ituple_A[0] = G;
        for (x = 0; x < G; x++) for (j = 0; j < N; j++)
            ituple_A[x*N+j+1] = A[ix+x][j];
            sprintf(tpname,"A%d\0",ix);
            status = cnf_tsput(tsd, tpname, ituple_A, tplength);
            ix += G; }

    // Wait for results
    received = i = 0;
    tplength = (1+N*N)*sizeof(double);
    if ((otuple = (double *)malloc(tplength)) == NULL) exit(1);
    while (received < N) {
        strcpy(tpname,"*");
        tplength = cnf_tsget(res, tpname, otuple, 0);
        G = (int) otuple[0];
        ix = atoi(tpname);
        iy = 1;
        // Assemble results in C
        for (i= 0; i < G; i++) {
             received ++;
             for (j=0; j < N; j++) C[ix][j] = otuple[iy++];
             ix ++; }}
    free(otuple);
    // Send the termination
    tplength = sizeof(double);
    if ((ituple_A = (double *)malloc(tplength)) == NULL) exit(1);
    ituple_A[0]  = -1;
    sprintf(tpname, "A%d\0",N*N);
    status = cnf_tsput(tsd, tpname, ituple_A, tplength);
    t1 = wall_clock() - t0;
    cnf_term();
}
```

Figure A.12: MatrixMaster.c for Synergy

```
/*-----------------------------------------------------
Matrix multiplication worker program.
-----------------------------------------------------*/
#include <stdio.h>
#include <synergy.h>
char tpname[20];
double ituple_B[N][N];
double *ituple_A; /* [0] = grain size, [1] = values per rows ... */
double *otuple;   /* [0] = grain size, [1] = values per rows ... */
double ita[N/2][N];
double ott[N/2][N];

main()
{
    int G, tsd, res, i, j, k;
    int ix, ia, ib, tplength;

    // Open Tuple Space Objects
    tsd = cnf_open("problem",0);
    res = cnf_open("result",0);

    // Ready to retrive Matrix B
    strcpy(tpname,"B*");
    status = cnf_tsread(tsd, tpname, (double *)ituple_B, 0);
    tplength = (1+(int)N*N/2)*sizeof(double);
    if ((ituple_A = (double *)malloc(tplength)) == NULL) exit(-1);
    while (1)
    {
        strcpy(tpname,"A*");
        tplength = cnf_tsget(tsd, tpname, ituple_A, 0);
        ix = atoi(&tpname[1]);
        if (tplength > 0) { // normal receive
            G = (int) ituple_A[0];
            // check for termination
            if (G == -1) {  // Send back so others can terminate
                status = cnf_tsput(tsd, tpname, ituple_A, tplength);
                cnf_term();
                return;
            }
            // Translate the A group
            for (i = 0; i < G; i++)
            for (j = 0; j < N; j++) {
                 ita[i][j] = ituple_A[i*N+j+1];
                 ott[i][j] = 0; // Initialize matrix C
            }
            // Dig (with optimized nesting order: (i,k,j))
            if ((otuple = (double *)malloc(tplength)) == NULL) exit(-1);
            otuple[0] = ituple_A[0];
            for (i =0; i < G; i++)
            for (k =0; k < N; k++)
            for (j =0; j < N; j++)
                ott[i][j] = ott[i][j] + ita[i][k] * ituple_B[k][j];
            // Assemble result tuple
            for (i = 0; i < G; i++)
            for (j = 0; j < N; j++)
                otuple[i*N+j+1] = ott[i][j];
            // Send result
            sprintf(tpname,"%d\0",ix);
            cnf_tsput(res, tpname, otuple, tplength);
            free(otuple);
        } else cnf_term();  // Done
    }
}
```

Figure A.13: MatrixWorker.c for Synergy (with CPU kernel)

```
/******************************************************************************
* Matrix Multiplication Program
* Heshan Suriyaarachchi
* Revised by: JYS
******************************************************************************/

#include <stdio.h>
#include "mpi.h"
// JYS: NRA, NCA and NCB are defined at compile time
#define MASTER 0     /* taskid of first task */
#define FROM_MASTER 1 /* setting a message type */
#define FROM_WORKER 2 /* setting a message type */

MPI_Status status;

double a[NRA][NCA],  /* matrix A to be multiplied */
       b[NCA][NCB],       /* matrix B to be multiplied */
       c[NRA][NCB];      /* result matrix C */

main(int argc, char **argv)
{
int numtasks,/* number of tasks in partition */
    taskid,     /* a task identifier */
    numworkers,/* number of worker tasks */
    source,     /* task id of message source */
    dest,      /* task id of message destination */
    nbytes,       /* number of bytes in message */
    mtype,        /* message type */
    intsize,/* size of an integer in bytes */
    dbsize,     /* size of a double float in bytes */
    rows,                      /* rows of matrix A sent to each worker */
    averow, extra, offset,      /* used to determine rows sent to each worker */
    i, j, k,/* misc */
    count;

struct timeval start, stop;

intsize = sizeof(int);    dbsize = sizeof(double);

MPI_Init(&argc, &argv);  MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks); numworkers = numtasks-1;

/*---------------------------- master ----------------------------*/
if (taskid == MASTER) {
  printf("Number of worker tasks = %d\n",numworkers);
  for (i=0; i<NRA; i++) for (j=0; j<NCA; j++) a[i][j]= i+j;
  for (i=0; i<NCA; i++) for (j=0; j<NCB; j++) b[i][j]= i*j;

  gettimeofday(&start, 0);

  /* send matrix data to the worker tasks */
  averow = NRA/numworkers; extra = NRA%numworkers;
  offset = 0;
  mtype = FROM_MASTER;
  for (dest=1; dest<=numworkers; dest++) {
    rows = (dest <= extra) ? averow+1 : averow;
    MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    count = rows*NCA;
    MPI_Send(&a[offset][0], count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    count = NCA*NCB;
    MPI_Send(&b, count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    offset = offset + rows;}

  /* wait for results from all worker tasks */
  mtype = FROM_WORKER;
  for (i=1; i<=numworkers; i++) {
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    count = rows*NCB;
    MPI_Recv(&c[offset][0], count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);}
  gettimeofday(&stop, 0);
  }  /* end of master section */
```

Figure A.14: Matrix Master for OpenMPI

```
/*---------------------------- worker (slave)----------------------------*/
if (taskid > MASTER) {
  mtype = FROM_MASTER;
  source = MASTER;
  MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
  MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
  count = rows*NCA;
  MPI_Recv(&a, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
  count = NCA*NCB;
  MPI_Recv(&b, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
  // JYS: Added Initialization
  for (i=0; i<NCB; i++}
    for (j=0; j<rows, j++)
      c[i][j] = 0.0;
  // JYS: Revised for optimized locality
  for (k=0; k<NCB; k++)
    for (i=0; i<rows; i++) {
      // JYS: Removed for better locality: c[i][k] = 0.0;
      for (j=0; j<NCA; j++)
        c[i][k] = c[i][k] + a[i][j] * b[j][k];
      }
  //mtype = FROM_WORKER;
  //MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&offset, 1, MPI_INT, MASTER, FROM_WORKER, MPI_COMM_WORLD);
  //MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, MASTER, FROM_WORKER, MPI_COMM_WORLD);
  //MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, FROM_WORKER, MPI_COMM_WORLD);
  }  /* end of worker */
  MPI_Finalize();
} /* of main */
```

Figure A.15: Matrix Worker for OpenMPI (with CPU kernel)