Sustainable GPU Computing at Scale

Justin Y. Shi, Moussa Taifi, Abdallah Khreishah, and Jie Wu
Dept. of Computer & Info. Sciences
Temple University
Philadelphia, PA 19122
$\{$*shi,moussa.taifi,akhreish,jiewu*$\}$*@temple.edu*

***Abstract*—General purpose GPU (GPGPU) computing has produced the fastest running supercomputers in the world. For continued sustainable progress, GPU computing at scale also need to address two open issues: a) how increase applications mean time between failures (MTBF) as we increase supercomputer's component counts; and b) how to minimize unnecessary energy consumption. Since energy consumption is defined by the number of components used, we consider a sustainable high performance computing (HPC) application can allow better performance and reliability at the same time when adding computing or communication components. This paper reports a two-tier semantic statistical multiplexing framework for sustainable HPC at scale. The idea is to leverage the powers of statistic multiplexing to tame the nagging HPC scalability challenges. We include the theoretical model, sustainability analysis and computational experiments with automatic system level multiple CPU/GPU failure containment. Our results show that assuming three times slowdown of the statistical multiplexing layer, for an application using 1024 processors with 35% checkpoint overhead, the two-tier framework will produce sustained time and energy savings for MTBF less than 6 hours. With 5% checkpoint overhead, 1.5 hour MTBF would be the break even point. These results suggest the practical feasibility for the proposed two-tier framework.**

***Keywords*-Fault tolerant GPU computing; Data parallel processing; Tuple switching network; Semantic statistical multiplexing;**

I. Introduction

The fastest supercomputers today use large number of GPU cards. The recent Chinese supercomputer, Tianhe-1, used approximately 7000 GPU cards to produce 2.5 petaflops performance [1]. Even faster machines are under construction. For example, the Oak Ridge National Laboratory plans to develop a new peta-scale supercomputer using an even larger number of "Fermi" GPUs to achieve an expected peak performance of 20 petaflops [2].

There are also two less publicized factors: the fast shrinking application mean time between failure (MTBF) [3] and fast growing energy consumption. Since a failed application must be restarted from the last checkpoint, optimal checkpoint interval can eliminate unnecessary energy waste. It is well known in the research community that both issues must be tamed for sustainable extreme scale computing.

With the current parallel processing environments, it is commonly accepted that to achieve higher performance, reliability must be sacrificed [4]. For higher application reliability, performance must be sacrificed. The interconnection network is the performance and reliability bottleneck that higher number of computing nodes can worsen application performance. Removing this bottleneck is considered very difficult.

Research finds that more sustainable architectures do exist with far less scalability constraints. For example, the packet switching networks [5] have delivered all-around scalability for many decades. Up scaling computing or communication components enables delivering higher service performance, higher data and service availability and less data/service losses. Although it is not immediately clear how the HPC applications could leverage the packet switching (statistical multiplexing) concepts, the scalability of the data networking architecture should be an important reference point for any computing/communication system at scale.

Inspired by the packet switching principles, we find that like a lost data packet, a lost computation can always be re-calculated if given the same input(s). This is true semantically for both deterministic and non-deterministic computations. Therefore, for HPC applications, spatial redundancy (redundant processing in parallel) is unnecessary. Since every parallel application employs many parallel "workers", these workers share the same code base, if each such task can be represented in a single "tuple", theoretically sustainable parallel processing would also be possible if we have a proper API and construct a statistically multiplexed high-level "tuple switch network".

This paper reports a two-tier HPC computation framework. In addition to a lower tier of traditional parallel tasks, we propose a statistically multiplexed interconnection network layer using the tuple space [6] abstraction. The idea is to leverage the powers of statistic multiplexing to meet the long standing HPC scalability challenges. Since packet processing is done "statelessly", the proposed framework is essentially a stateless parallel processing (SPP) machine.

This paper is organized as follows: Section 2 is a brief survey of existing fault tolerant parallel computing methods. Section 3 describes the technical motivating factors of the proposed methodology. Section 4 contains theoretical discussions on *push* and *pull* parallel processing paradigms in performance and sustainability perspectives. Section 5 introduces stateless parallel processing and its sustainability analysis. Section 6 reports the design and results of computational experiments. Section 6 contains the discussion of the results. Section 7 contains the summary and future directions.

II. Fault Tolerant Computing

Component failure in a computer is a small probability event. When the component count is large, however, the cumulative effects are formidable. The lack of all-around scalability in existing HPC applications has quickly driven their MTBFs from weeks down to 60 minutes [3]. This means that it would soon be impossible to have a full hour continuous run for a large scale HPC application using the current combination of hardware and software.

To preserve the valuable intermediary results, checkpoint-restart (CPR) [7] 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 could restart the application from the last checkpoint, thus preserving the energy that had generated the saved results.

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 environment that allows the application program to call for a checkpoint using a single instruction. Recovery is automatic. The Berkeley's BLCR library [8] is an example of system level CPR. Application level CPR [9] 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 automatic recovery after failure.

In a typical GPGPU parallel system, the host processor must use shared memory to communicate with multiple GPU cards and a system level checkpoint must save the memory contents shared amongst all processors. At the present time, this is considered a non-trivial challenge ([10], [11] and [12]).

CheCUDA [10] reported one method that can save the state of a single GPU for a later restart or migration. A number of improvements were also suggested to reduce its large overheads. Authors of [11] reported an effort using a virtualized GPU. [12] reported a method using a stream interface provided by the NVIDIA CUDA to hide latency. At the time of this writing, the CheCUDA is still unstable. Due to the low level memory interface complexity between multiple vendors, system level CPR for multiple GPGPU is still under development. Future fused CPU-GPU architectures may lesson the difficulties. But the current interests are in portable computer productions. The use of fused CPU-GPU processors in supercomputers is still on the drawing board.

Existing HPC programming frameworks are primarily based on the message passing standards [13], such as OpenMPI [14], MPICH [15] and MVAPICH [16]. All these systems provide an API for check-pointing the state of individual nodes using the BLCR library and and an API for communication channel check-pointing using a variety of techniques [9].

In practice, 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 ([17], [18]). For general purpose HPC, fault tolerance at scale is considered very difficult [19].

III. Motivation

The unique architectural advantage in packet switching protocol design is the primary motivation of the reported research. This unique architecture allows unlimited scalability in performance and reliability as the network component count increases. Since all HPC applications must involve communications, we paid specific attention to the timeout treatments in HPC applications. Interestingly, we found that most people believe that a message timeout is identical to a fatal error. Therefore, the application must halt.

The timeout of a communication request really means the request's state is unknown. Treating it as a fatal error forces the entire application to halt on every possible transient component error. Thus, as the component count increases, the application's MTBF decreases. This is the root cause of HPC application sustainability challenges.

In contrast, the lower level data networks treat timeout in fundamentally different ways. In packet switching networks, a re-transmission logic coupled with idempotent processing has been proven a winning formula if supported by a statistically multiplexed infrastructure. The counter-intuitive packet switching protocols have delivered practical sustainable data networks for many decades. It is perhaps the most scalable man-made architectures in human history.

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

- 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 network based on the unit of transmission.

Without statistical multiplexing, the actual state of a communication task is theoretically not confirmable ([20], [21] and [7]). With statistical multiplexing, the probability of success increases proportionally as the number of redundant path increases. The re-transmission protocol implements transient storages for the transmission units. Therefore, increasing the networking component counts improves the

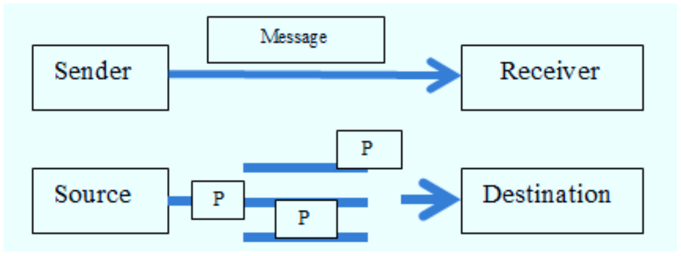

Figure 1. Messaging network and packet-switching data network.

network's collective performance and reliability at the same time. Although the packet-switching overhead is significant compared to direct circuit-switching protocols, the low cost fault tolerance and unlimited scalability of packet switching concept have been proven to prevail in practice.

One often wonders why the low level data network benefits are not automatically inherited by higher level applications. The answer is that they operate on different units of transmissions (Figure 1). In high level applications, the data objects, such as messages, are only transmitted once. Since the mutual information is zero between the protocol layers [22], by the end-to-end principle [23], not only a statistically multiplexed infrastructure is necessary, but also the application programming interface (API) must include the essential elements of the packet-switching principle for the application to be sustainable.

Therefore, HPC application sustainability naturally happens if the units of computations are statistically multiplexed. We propose a robust runtime statistically multiplexed "tuple switching network" [24] and a high-level "tuple-driven" parallel processing API in order to contain the increasing risks of massive component failures.

IV. Tuple-Driven Parallel Processing

In a "tuple-driven" parallel processing environment, automatic worker failure protection can be provided by allowing a retrieved tuple of a work assignment 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 [17]. In view of the "tuple switching network", this mechanism satisfies statistical multiplexing requirement analogous to the automatic re-transmission of TCP packets.

In other words, the very feature that was responsible for putting the data parallel processing methods into the "Quasimodo" rank is actually indispensable for larger scale computing.

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 GPGPUs are exclusively used for parallel workers, this means that we could use data parallel programming to overcome the system level multiple GPU checkpoint challenges.

Unlike explicit parallel programming methods, tuple-driven parallel programming relies on communicating tuples to automate task activation (also called "firing" in literature [25]). This enables automatic formation of SIMD, MIMD and pipeline clusters at runtime. The net-benefit of this feature is automatic "hiding" of communication latencies (or "work stealing" [26]). For extreme scale HPC applications, these sustainable qualities are desirable.

In addition to "stateless" workers, a practical HPC application must also contain "stateful" masters (programs responsible for delivering the semantically identical results as sequentially computing the same data). The masters must still be check-pointed 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 inherent inefficiencies of tuple parallel processing.

A *pull*-application programming environment forces the programmers to focus on data partitions (or exposing parallelism). The data partitioning strategy determines the ultimate deliverable performance.

The *pull*-based API does not have a fixed process-data binding. Therefore, it is possible to statistically multiplex the higher level data contents. Specifically, if the *pull*-based API contains re-transmission and idempotent processing for HPC computing tasks, all-around scalability should be certainly attainable.

The *pull*-applications require the processing environment to support a data repository for matching computing tasks at runtime. This introduces an additional communication overhead that almost doubles the overhead for every direct inter-processor communication request.

The increased overheads allow the introduction of statistical multiplexing on the high level HPC semantic network:

1) The potential to deploy multiple interconnection networks in parallel, thus relieving the performance bottleneck to allow more computing nodes in parallel and to support diverse communication patterns.
2) The potential to offer automatic worker fault tolerance, thus reversing the negative effect on application MTBF and delivering sustainable performance with automatic failure containment.

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

Like the Internet, a *pull*-based two-tier framework can potentially deliver scalable performance, scalable availability, scalable service losses and energy efficiency at the same time.

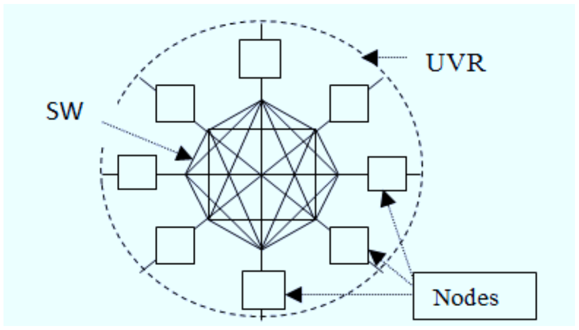

Figure 2. Tuple Switching Network.

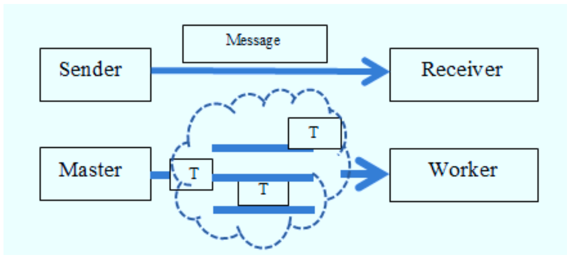

Figure 3. Static and statistical semantic networking.

The following sections reports findings based on an experimental *pull*-based statistical multiplexing project named Synergy [27].

V. Stateless Parallel Processing

A. Architecture

Stateless parallel processing (SPP) [24] was inspired by the sustainability of packet switching architecture for data networks. Figure 2 shows the conceptual diagram of the proposed "tuple switching network".

In Figure 2, SW represents a collection of redundant network switches. 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 multiple GPU cards.

A global tuple space is implemented as follows:

1) Data requests travel through a UVR.
2) All nodes participate in data matching in parallel.
3) All networks participate in direct data exchanges.

These functions are implemented in a single daemon that runs on each node. Like the peer-to-peer file sharing systems, the SPP daemons communicate with each other to form a single consistent HPC machine image using any available resources. Each daemon holds the local data. Data matching requests travel by UVR. The actual data transfers are done in parallel via the multiple redundant physical networks. For parallel applications with optimized grain sizes, only a few data items should reside on each node. At application level, all nodes participate in a statistically multiplexed global tuple space. Applications use the tuple space API to communicate with local daemon which in turn communicates with other daemons to complete data acquisition in parallel. Each application exploits multiple redundant networks automatically to counter-balance the speed disparity between computing and communication. 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 CPU and GPU 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 overall scalability: adding computing nodes or networks will increase both application's performance and reliability.

Figure 3 makes a conceptual comparison between the static (*push*-application) and statistical multiplexed semantic networks (for SPP applications), where T stands for "tuple" which is the unit of transmission of the semantic network.

In Figure 3, the *push* API does not contain re-transmission and idempotent processing of units of transmissions. Each message is like a UDP packet in data networks, it only gets sent once. The tuples in the SPP semantic networks are like TCP virtual circuits with automatic re-transmission and idempotent processing built-in.

B. Application Development

A *pull*-application will use only data manipulation commands. The Tuple Space abstraction [6] naturally fits our needs.

The tuple space API contains three data manipulation primitives [24]:

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.

The "&" sign represents "access by reference" convention 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 should 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

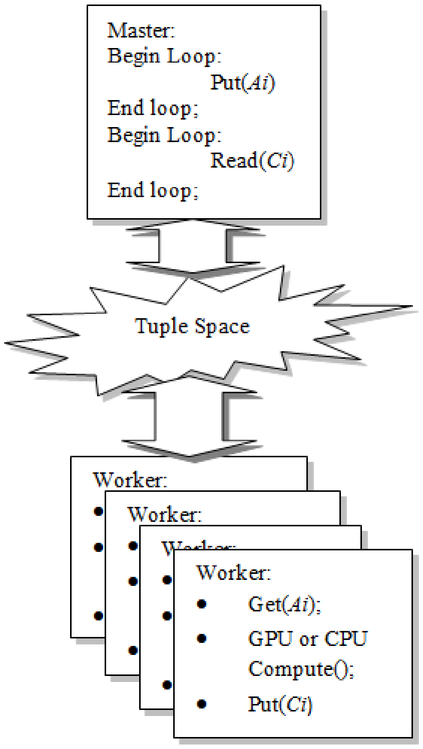

Figure 4. Logical View of SPP Programming.

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 processing environments, it will run on all nodes authorized by the application owner. The application terminates when there are no more assignment tuples.

The tuple operations must be implemented with the exact semantics as specified. Implementing tuple space using compiler generated codes, such as the approach taken by the Linda project [6], makes it impossible to use statistical multiplexing.

C. Sustainability Analysis

In this section, we assess the expected time savings using "worker fault tolerance", as promised by the SPP statistical multiplexing framework.

To do this, we build two models (based on [17]) for a typical HPC application with check-points. The first one is for *push*-based parallel programming systems where any component failure would cause the entire application to halt. The second is for SPP 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 [18] where the optimal check-point interval model was based on a system exhibiting Poisson single component failures, the following models assume multiple Poisson component failures.

According to [17], we define the expected computing time with failure, as follows:

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

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 check-point 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 check-point 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. 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 check-point 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 check-point interval with worker fault tolerance is:

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

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 delay $X = 30$ sec $= 0.5$ minute, the expected time 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:

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

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 [28].

In other words, for an application that needs 30 hours computing time, the total savings would be about 5 Megawatt hours (1024 processors with 187 watts per processor).

Assuming the SPP (Synergy) slow down factor = 3, Figure 5 shows that 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. Figure 5 shows that even with three-times slower performance, the SPP frame work still delivers sustained performances. Higher CPR overheads accelerate the benefits.

In practice, these figures suggest the break even points between using single tier mono-scale simulation to two-tier, possibly multi-scale simulation. Since GPUs are exclusively used for workers, SPP offers system level multiple GPU fault tolerance without involving check-pointing GPU/CPU shared memories.

VI. Computational Experiments

A. Experiment Setup

Application. 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: $(0 < i \leq k)$:

$$C = A_k \times B,$$
$$A_i = A_{i-1} \times B$$

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.

Due to MPI programming limitations, processing granularity is always $\frac{N}{P}$ (P=number of processors) and cannot be adjusted after compilation. Synergy does not have this limitation.

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) Running times without check-points.
2) Checkpoint overhead.
3) Performances with check-points without failure.
4) Performances with check-points and recovery with failure injections.

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 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). (http://www.ncsa.illinois.edu/UserInfo/Training/Workshops/CUDA/) and Intel C++ compiler 10.0 for Linux (http://www.ncsa.illinois.edu/UserInfo/Resources/Software/Intel/Compilers/10.0/C_Release_Notes.htm).

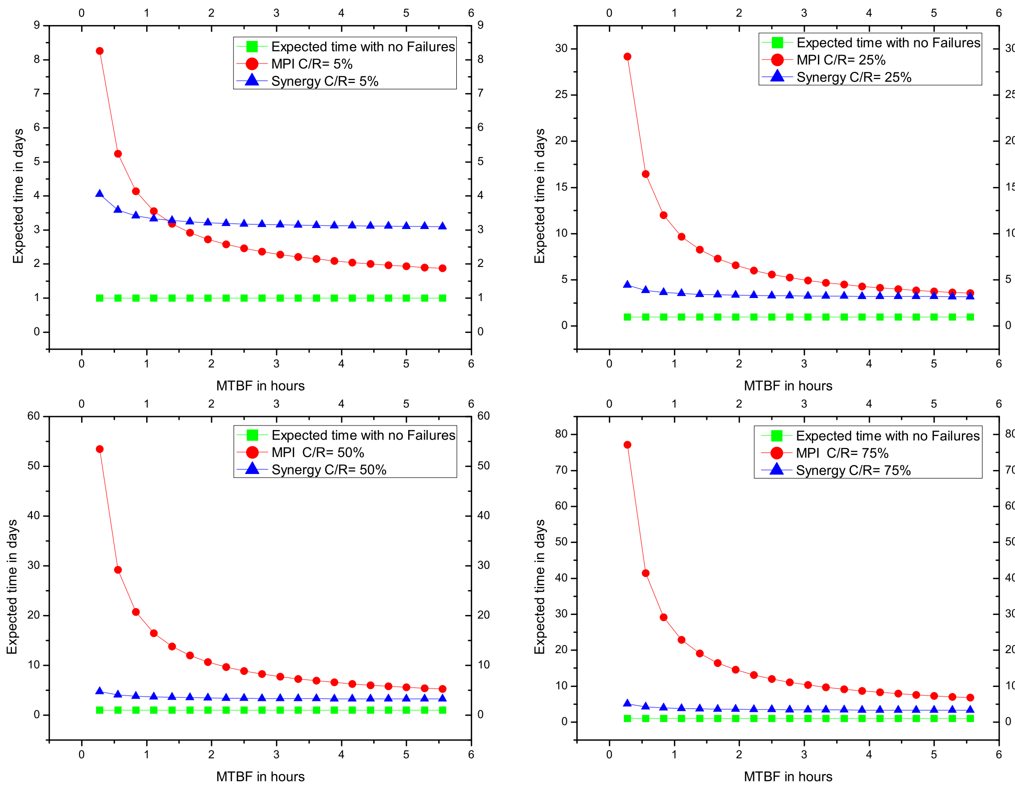

Figure 5. Sustainability analysis: Expected Elapsed Time vs. MTBF (P=1024, Synergy Overhead=3)

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 [29], to perform the injection of failures. The failure injection algorithm accepts variable MTBFs.

B. Computation Experiments and Results

In practice, the check-point creation time K_0 is different for MPI and for Synergy. For MPI, the check-point must include the global state of all involved masters. For Synergy, the master check-point only needs to include local states. Multiple masters will check-point in sync and in parallel using a distributed synchronized termination algorithm [30].

The recovery time K_1 would also be different. For MPI, the recovery time is a simple reading of a globally saved state. For Synergy, the recovery time includes launching multiple masters reading the saved states in parallel. For simplicity, we consider the differences negligible.

In the reported experiments, we used only a single master.

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 MPI implementation produces the best results at 19 GPU workers where the granularity is at 600. The same figure shows that the Synergy application does best with 5 GPU workers and a granularity of 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 use workers that are solely geared toward finding a GPU device, 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 (worker fault tolerance [24]).

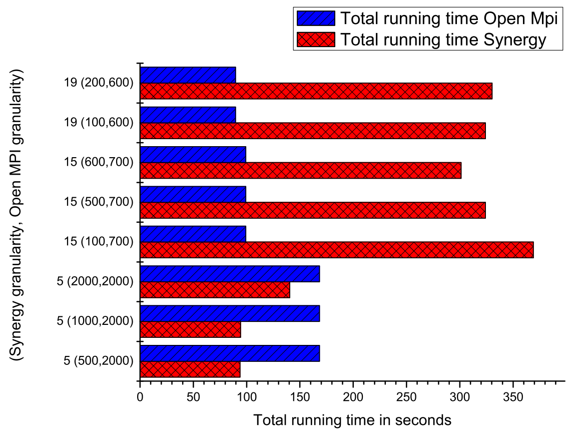

Figure 6. Performance without checkpoint and failure (N=10,000, Rounds = 10).

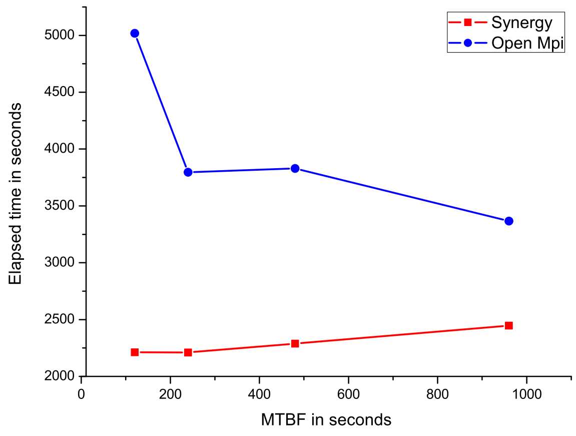

Figure 7. Running time with failures (N=10,000, Rounds=400, P=5(Synergy), P=19(OpenMPI).

The reported computation results were recorded with the following parameters:

- $N = 10,000$.
- $P = 5$ or $P = 19$ (adjusted for optimality).
- $K = 400$ (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 the processing nodes statistically.

Figure 6 shows the performance differences between OpenMPI and Synergy without checkpoints. 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, since there was only a single interconnection network in the test environment. For the same amount of work, MPI program needed more GPU workers.

As mentioned earlier, our application level check-point 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 optimal intervals according to the discussions in Section 5.3.

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 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 5.3). In this setup, the master receives more than expected failures when $P = 1024$.

Figure 7 shows the computational results for OpenMPI and Synergy with simulated failures.

In Figure 7, we used accelerated MTBFs to demonstrate the net effects of multiple CPRs within the context of the experiment setup. These results are consistent with the sustainability model predictions.

VII. Discussions

This paper proposed a two-tier framework for sustainable HPC applications at scale. We argued that an Internet-like strategy is necessary in order to mitigate the increasing risks of massive component failures and to deliver lasting sustainability for HPC at scale.

We proposed the need for a tuple-switching network based on the recognition of the dynamic semantic network of a running HPC application. We showed that *push*-based parallel applications have fragile semantic network, it is not possible to leverage statistic multiplexing to counter scalability challenges. We propose to measure sustainability when adding computing and communication components for simultaneous benefits in

1) increased application performance,
2) increased application's availability, and
3) reduced computation loses.

We have shown that one possible framework to deliver the above measures is the SPP architecture using the concept of statistical multiplexing of application's semantic network.

Sustainable HPC also brings energy efficiency. The savings come from drastically reduced global checkpoints. There are also more potentials for productivity gains. For example, it may not be necessary to manually write the "wrappers" to include legacy applications into an extreme scale application. Automated tools have been experimented with using Parallel Markup Language (PML)[31]. More efforts are needed to study how to compose multi-scale codes using the proposed tuple space parallel programming environment [32]. This also applies to the GPGPU applications that dedicated personnel would focus on producing the optimized GPU kernels while the wrapper would be automatically generated.

HPC application batch scheduling would be trivialized since the SPP applications can exploit any available resources and optimization is built-in. Non-stop HPC would become a reality where components can be taken offline for repair without shutting down the running applications. Energy 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 becomes a reality, optimization models are also needed to help users calculate the optimal strategy to maximize the yield of a given budget for each computing intense application. Theoretical models are also needed to study the stability of even larger systems that integrate physical sensors and wireless components (cyber physical systems).Since the tuple switching network is very similar to the packet switching network, with the recent hardware advances in high performance FPGA circuits [33], it is possible to develop direct hardware support for HPC and cyber physical systems needs to further improve the overall performances for all applications.

VIII. Conclusions

The fundamental result of the reported research is the use of statistical multiplexing to solve the seemingly "impossible" computational problems. We have shown that the powers of statistic multiplexing can indeed tame the nagging HPC sustainability challenges that have troubled us for a long time. Our limited computational experiments showed the practical feasibility that confirms with the greater implications described in the theoretical models. We showed that extreme scale HPC can be practical via the proposed two-tier framework.

Although the proposed two-tier framework can potentially up scale to extreme large sizes, the actual deliverable performance for any given application is still confined by the maximal available and exploitable resources at runtime. The proposed framework merely removes the structural impediments. Semantic multiplexing differs from the "messaging-switching network" concept [34] in the recognition of different units of transmissions. Like the packet switching protocol that has delivered stochastic and reliable deterministic data service for many decades, we expect the same benefits for higher level semantic networks. Optimization of semantic network multiplexing is a new research problem since most higher level semantic networks are implemented using the TCP/IP protocol. Optimization schemes are needed to eliminate unnecessary redundancies.

The reported result has far-reaching consequences. Statistic multiplexing of semantic network can also solve sustainability problems for other applications, such as online transaction processing, storage networks and service oriented architectures [28]. It can theoretically eliminate all communication-induced uncertainties – a desirable feature for all mission critical applications. Since most applications will naturally gravitate towards mission critical status, the concept of statistical multiplexing of semantic network is 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] Wikipedia, "Fastest chinese supercomputer." 2009, [Online], http://en.wikipedia.org/wiki/Tianhe-I.

[2] HPCwire, "ORNL looks to NVIDIA Fermi architecture for new supercomputer." 2009, [Online], http://www.hpcwire.com/offthewire/ORNL-Looks-to-NVIDIA-Fermi-Architecture-for-New-Supercomputer-62919517.html.

[3] F. Cappello, "Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities," *Int. J. High Perform. Comput. Appl.*, vol. 23, pp. 212–226, August 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1572226.1572229

[4] "HPC resilience consortium," 2010, [Online], http://resilience.latech.edu.

[5] P. Baran, "On distributed communications, rm-3420," http://www.rand.org/about/history/baran.list.html, Tech. Rep., 1964.

[6] N. Carriero and D. Gelernter, *How to Write Parallel Programs - A First Course*. Cambridge, MA: The MIT Press, 1990.

[7] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[8] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006. [Online]. Available: http://stacks.iop.org/1742-6596/46/i=1/a=067

[9] M. Shultz, G. B. R. Fenandes, D. M. K. Pingali, and P. Stodghill, "Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs," in *Proceedings of Supercomputing 2004 Conference*, Pittsburgh, PA., November 2004.

[10] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "Checuda: A checkpoint/restart tool for cuda applications," in *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, dec 2009, pp. 408–413.

[11] S. Laosooksathit, C. Leangsuksan, A. Dhungana, C. Chandler, K. Chanchio, and A. Farbin, "Lightweight checkpoint mechanism and modeling in GPGPU," in *Proceedings of the hpcvirt2010 conference*, 2010.

[12] Wikipedia, "CUDA-compute unified device architecture," 2010, [Online], http://en.wikipedia.org/wiki/CUDA.

[13] H. Rolf, "The MPI standard for message passing," in *High-Performance Computing and Networking*, ser. Lecture Notes in Computer Science, W. Gentzsch and U. Harms, Eds. Springer Berlin / Heidelberg, 1994, vol. 797, pp. 247–252, 10.1007/3-540-57981-8-126. [Online]. Available: http://dx.doi.org/10.1007/3-540-57981-8-126

[14] 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, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, D. Kranzlmller, P. Kacsuk, and J. J. Dongarra, Eds. Springer Berlin / Heidelberg, 2004, vol. 3241, pp. 353–377, 10.1007/978-3-540-30218-6-19. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30218-6-19

[15] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *Supercomputing, ACM/IEEE 2002 Conference*, nov. 2002, p. 29.

[16] M. Koop, T. Jones, and D. Panda, "Mvapich-aptus: Scalable high-performance multi-transport MPI over infiniband," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1–12.

[17] W. Gropp and E. Lusk, "Fault tolerance in message passing interface programs," *Int. J. High Perform. Comput. Appl.*, vol. 18, pp. 363–372, August 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1080704.1080714

[18] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, p. 303C312, 2006.

[19] E. N. Elnozahy and J. S. Plank, "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, pp. 97–108, April 2004. [Online]. Available: http://dx.doi.org/10.1109/TDSC.2004.15

[20] M. Herlihy and N. Shavit, "The topological structure of asynchronous computability," *J. ACM*, vol. 46, pp. 858–923, November 1999. [Online]. Available: http://doi.acm.org/10.1145/331524.331529

[21] A. Fekete, N. Lynch, Y. Mansour, and J. Spinelli, "The impossibility of implementing reliable communication in the face of crashes," *J. ACM*, vol. 40, pp. 1087–1107, November 1993. [Online]. Available: http://doi.acm.org/10.1145/174147.169676

[22] T. Cover and J. Thomas, *Elements of Information Theory*. John Wiley & Sons, Inc., 1991.

[23] J. Saltzer, D. Reed, and D. Clark, "End-to-end arguments in system design," April 1981, pp. 509–512.

[24] J. Y. Shi, "Decoupling as a foundation for large scale parallel processing," in *Proceedings of 2009 High Performance Computing and Communications*, Seoul, Korea, 2009.

[25] J. B. Dennis, "Data flow supercomputers," *Computer*, pp. 48–56, 1980.

[26] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other cilk++ hyperobjects," *ACM Symposium on Parallelism in Algorithms and Architectures*, 2009.

[27] Y. Yijian, "Fault tolerance protocol for multiple dependent master protection in a stateless parallel processing framework," Ph.D. dissertation, Temple University, August 2007.

[28] J. Y. Shi, *Chapter 5:Fundamentals of cloud application architectures, Cloud computing: methodology, system, and applications*. CRC, Taylor & Francis group, 2011.

[29] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*, 2nd ed. McGraw-Hill Higher Education, 1997.

[30] B. Szymanski, Y. Shi, and N. Prywes, "Synchronized distributed termination," *IEEE Transactions on Software Engineering*, vol. SE11, no. 10, pp. 1136–1140, 1985.

[31] F. Sun, "Automatic program parallelization using stateless parallel processing architecture," Ph.D. dissertation, Temple University, 2004.

[32] "Multi-scale modeling and simulation," 2010, [Online], http://www.math.princeton.edu/multiscale/.

[33] "High performance FPGA development group," 2010, [Online], http://www.fhpca.org/.

[34] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, July 2000, ch. 3.7 TCP Congestion Control.