FOX: A Fault-Oblivious Extreme-Scale Execution Environment
FY13 Progress Report

FOX team

LLNL, Sandia CA, PNNL, Boston U., Ohio State U., Bell Labs

Exascale computing systems will provide a thousand-fold increase in parallelism and a proportional increase in failure rate relative to today's machines. Systems software for exascale machines must provide the infrastructure to support existing applications while simultaneously enabling efficient execution of new programming models that naturally express dynamic, adaptive, irregular computation; coupled simulations; and massive data analysis in a highly unreliable hardware environment with billions of threads of execution.

The FOX project is developing systems software and runtime support for a new approach to the data and work distribution for fault oblivious application execution. Our OS work includes adaptive, application tailored OS services optimized for multi $\rightarrow$ many core processors. We have developed a new operating system NxM that supports role-based allocation of cores to processes. We have built a task queue library based on distributed, fault tolerant key-value store. We are developing fault tolerance mechanisms for task parallel computations employing work stealing for load balancing. We are developing a task parallel library based on the Linda tuple space model.

1 Operating Systems

1.1 NxM OS

Collaborators at Google set up a git-based repository hosted by coreboog.org to maintain NxM, a new version of the NIX operating system developed by Sandia CA in collaboration with Bell Labs, Vita Nuova, and the University of Rey Juan Carlos. The NxM repo provides code review, automatic compilation, and regression testing on top of the Gerrit framework. It is also possible to cross-compile the entire operating system from Linux or OS X for ease of development. Because the code was changed significantly in moving to the new environment, NIX was re-named to "NxM" to distinguish from the original codebase and development environment.

Work on NxM has focused largely on two areas:

- Support for running Linux processes at native speed
- iSCSI

Acknowledging the large body of existing Linux programs, we have added the capability to run Linux binaries on NxM. The NxM kernel interface (defined by its system calls) is not wholly incompatible with the Linux kernel interface expected by Linux programs. By adding some simple code to our kernel, we can translate Linux system calls into NxM calls; the most important calls (such as `read` and `write`) are implemented directly in the kernel, while less essential calls (such as `getrlimit`) are implemented in a userspace "shepherd" program (a program developed under the DoE-funded HARE effort which serves to load and launch the Linux program). This capability has been used to run sample applications such as miniFE, which ran at around 99% the speed of native Linux. Work continues to improve the efficiency of this capability, with the aim of potentially surpassing Linux performance.

We have also experimented with using NxM to provide I/O and storage capabilities, specifically iSCSI storage. I/O is an increasingly important part of HPC systems; for example, checkpoint/restore functionality requires massive disk storage throughput capabilities. Team members at Bell Labs have created a userspace iSCSI server which we are in the process of evaluating; while early tests look good, more complete and thorough tests have not yet been completed. Work is also underway on code that will allow us to shift userspace programs into the kernel at will, providing performance enhancements by reducing context switches. The iSCSI server, being an I/O-heavy component, should be a useful test of this capability.

1.2 Elastic Building Blocks

The Boston University team has developed a generalized model Elastic Building Blocks for developing future HPC applications that are hybrids of customized runtimes and general-purpose commodity operating systems. The model targets the development of libraries and applications that exploit distributed data structures and associated communication optimizations in the face of dynamic changes to the set of nodes. We have actively been defining this model and developing it in the context of constructing a prototype runtime for supporting hash-table software. Our goal is to complete a prototype of the runtime that supports a custom hash-table to provide an alternative implementation of libfox (see below). An outgrowth of this work has been a runtime model that uses HPC systems approaches along with distributed systems research techniques to yield an environment that combines commodity software stacks with reusable high performance software that reacts to dynamic changes.

In the last quarter, we have focused on system-level hash tables that can serve as building blocks for our implementation of libfox. In particular we have been focusing on the tension between high-performance and fault-tolerant distributed systems approaches.

- We have designed a total ordered broadcast [1] algorithm appropriate for Exascale systems. Specifically the latency of an total ordered broadcast message grows logarithmically with the number of nodes. A key property of the design is portability across differing interconnects. While an implementation could exploit hardware fault-tolerant collectives it does not depend on them and can be implemented strictly on top of point-to-point communication. Its fault-tolerant behavior assumes crash stop failures, the existence of failure detectors [2], and the absence of network partitioning. These assumptions are equivalent to those put forward by the MPI Fault Tolerance working group proposals [3]. Our algorithm maintains logarithmic latency for total ordered broadcasts independent of the number of failing nodes.

- We have continued to develop our library OS based runtime [4]. The goal of this runtime is to advance the standard HPC light-weight kernel approach to target the construction of new high-performance fault tolerant applications while still not requiring full OS support on the compute nodes. In particular we added support for C++ exceptions and runtime type identification to support a broader range of C++ software. Additionally we ported versions of the C and C++ standard libraries to our runtime. This work enables us to construct rich software compositions on a node while still not requiring a full OS. A core property of the runtime is the support for inter-node communications to this end we added support for the BG/P networks and addition ethernet based NICs.

- Building on the single node shared memory hash tables developed in prior quarters we prototyped some simple distributed hash tables on top of the raw RDMA facilities of BG/P. In this work we side stepped all MPI and higher level interfaces to provide use a building block for our next quarter's work.

Our goals for the remainder of the project are as follows.

(a) Publish our total order broadcast algorithm along with a Blue Gene implementation and evaluation.

(b) Integrate our total order broadcast, new runtime and distributed hash tables into a library OS implementation of libfox on Blue Gene and conduct an evaluation with the libfox apps.

1.3 FusedOS Integration

The IBM Research FusedOS project follows the core specialization theme by combining a Linux service core with multiple application-focused CNK cores. IBM Research rebased and updated the base BG/Q Linux kernel to make it more compatible to the Linux community. They removed unnecessary dependency on BG/Q firmware; moved thread/core startup to the simpler Kexec model, allowing the possibility of reboots without reconfiguration of the nodes; and added dynamic configuration through the device tree. This reorganization was done to increase the likelihood of adoption of the patches into the mainstream PPC Linux base.

2 Task Parallel Runtimes

2.1 Tuple Space Programming Model

We initially explored alternatives to the communicating sequential processes (CSP) programming model of MPI by replacing point-to-point and collective communication with key-value store puts and gets. The initial work with Memcached demonstrated two important features post-MPI programming models should exhibit:

- Processes are decoupled in space
- Processes are decoupled in time

To address faults, rigid assumptions on where and when data resides, where and when computation should take place, and where and when communication occurs are no longer possible. Memcached, however, was designed mainly for enterprise systems and as such exhibits certain design drawbacks for high performance scientific computing. Namely, we identified the following problems:

- Lack of compatibility with high-performance interconnects (sockets only; very difficult to adapt to, e..g, native RDMA transfers)
- High overheads and lack of zero-copy semantics for large data transfers.
- Lack of expressiveness. Key-value store interface does not readily express all the available parallelism to the compiler or runtime. Key-value gets/puts could also be somewhat tedious to use for communication.
- Inability to add generic event listeners for when certain key-value pairs become available.

Over the last few months, we have been developing a derivative of the Linda programming model which generalizes the key-value store to arbitrarily-typed tuples and even wildcard matching. A flexible C++ template library allows dependencies between tuples and tasks to be easily expressed. Arbitrary event listeners can be attached to tuple operations, allowing tasks to immediately respond as new data becomes available. Although the framework most naturally suggests a Task-DAG model, the tuple space framework is designed to be as "expressive" as possible, allowing multiple parallel models to be expressed.

A major reason for choosing a Linda-like approach is a large body of work on fault-tolerant Linda from the 1990s. In particular, Linda can be extended naturally with a a set of fault-tolerant transactions for resilient computation. In the shorter-term, our resilience experiments are focusing on "slow nodes" rather than totally failed nodes. This should demonstrate clearly the flexibility of the Linda runtime that derives from processes being decoupled in space and time.

The tuple space library has been completed. In addition, a highly-efficient tuple transport layer has been developed on top of the native Cray GNI network interface. Although not yet portable to, e.g., IBM systems, the transport layer explicitly takes advantage of network optimizations provided by modern NICs, including asynchronous RDMA transfers of large data and very low-latency short message mailboxes. We are therefore aiming for a "proof-of-principle" demonstration rather than a portable library.

2.2 Task Model based on Global Arrays: TASCEL

Load Balancing Tasks Using Work Stealing Dynamic load balancing is a promising technique to adapt to variations in the execution environment such as irregular computation, faults, system noise, and energy constraints. We have developed distributed memory load balancing algorithms based on work stealing and demonstrated scalability to hundreds of thousands of processor cores [5]. Our techniques are shown to incur low overheads (space and time) to ensure fault tolerance, with the overheads decreasing with per-process work at scale. We demonstrated consistently high efficiencies on ALCF Intrepid, NERSC Hopper, and OLCF Titan for the Hartree-Fock and Tensor Contraction benchmarks.

Based on the demonstration of scalable load balancing, we ported a homology detection framework on TASCEL [6]. The implementation employed distributed memory work stealing to effectively parallelize optimal pairwise alignment computation tasks. This implementation was evaluated on up to 131,072 cores of the Intrepid IBM BlueGene/P system.

The flexibility inherent in work stealing when dealing with load imbalance results in seemingly irregular computation structures, complicating the study of its runtime behavior. We developed an approach to efficiently trace async-finish parallel programs scheduled using work stealing [7]. We identify key properties that allow us to trace the execution of tasks with low time and space overheads. We also study the usefulness of the proposed schemes in supporting algorithms for retentive work stealing. We demonstrate that the perturbation due to tracing is low and the traces are concise, amounting to a few kilobytes per thread in most cases. We also demonstrate that the traces enable retentive stealing for recursive parallel programs.

Another direction of our work has focused on developing effective load balancing techniques via work stealing for heterogeneous parallel clusters with attached accelerators at cluster nodes. In many application contexts, including compute intensive couple cluster methods like CCSD(T), dynamic load balancing of tasks is a critical challenge. In addition to the fact that a given task has different execution times on the host CPU versus attached accelerator, the collection of parallel tasks is also non-homogeneous in that some tasks are faster to execute on one or more CPU cores, while others are much faster to execute on a GPU accelerator.

However, it is feasible to predict the relative execution times of any task on the CPU and GPU, based on a priori characterization as a function of task operand sizes. This key characteristic was exploited in developing and evaluating several alternative designs for an accelerator-aware, receiver-initiated work-stealing approach for dynamic load-balancing [8]. It was found that overlapping of computations on CPUs and GPUs with pipelined asynchronous PGAS data movement across the nodes of the cluster, as well as intra-node movement across the link between CPUs and GPUs, was critical to achieving high performance. For the CCSD(T) benchmark, on a 32-node heterogeneous cluster, an overall speedup of over 2.5X was achieved over either CPU-only execution or GPU-only execution.

Fault Tolerance Checkpointing approaches to fault tolerance that employ collective rollback typically revert all the processes to the previous checkpoint in the event of a failure, a heavyweight solution at exascale. We developed fault tolerance mechanisms for task parallel computations employing work stealing [9]. The computation is organized as a collection of tasks with data in a global address space. The completion of data operations, rather than the actual messages, is tracked to derive an idempotent data store. This information is also used to accurately identify the tasks to be re-executed in the presence of random work stealing. We consider three recovery schemes that present distinct trade-offs lazy recovery with potentially increased re-execution cost, immediate collective recovery with associated synchronization overheads, and noncollective recovery enabled by additional communication. We employ distributed-memory work stealing to dynamically rebalance the tasks onto the live processes. We demonstrate that the overheads (space and time) of the fault tolerance mechanism are low, the costs incurred due to failures are small, and the overheads decrease with per-process work at scale.

3 Applications

Graph traversal Our asynchronous task-queue version of the Graph500 benchmark has been ported to BG/P and has been tested at scale on 131k cores. It currently performs within 20% of the current Graph500 Intrepid results. Over the next quarter we will work to improve the task-queue performance, and explore other graph algorithms such as connected components and weighted shortest paths. In the process of porting our Graph500 benchmark, we have developed scalable algorithms to generate R-MAT and Preferential Attachment scale-free graphs on 32k nodes of Intrepid.

We have implemented two new graph algorithms using the asynchronous task-queues. We have developed new versions of Kth-core decomposition and Triangle counting, and tested them at scale on Intrepid using our INCITE award. This work will be published in IPDPS 2013.

Tuple space benchmarks With only a minor delay (1 month) to the original milestone of demonstrating a chemistry application at scale, we expect to perform scaling tests of the Fox-Linda programming model on example problems including systolic matrix multiplication, a finite element mini-app, and a molecular dynamics chemistry mini-app. Debugging efforts on the distributed memory matrix multiplication tests are already underway.

References

[1] X. Défago, A. Schiper, and P. Urbán, “Total order broadcast and multicast algorithms: Taxonomy and survey,” *ACM Comput. Surv.*, vol. 36, pp. 372–421, Dec. 2004.

[2] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *J. ACM*, vol. 43, pp. 225–267, Mar. 1996.

[3] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, “An evaluation of user-level failure mitigation support in mpi,” in *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, EuroMPI’12, (Berlin, Heidelberg), pp. 193–203, Springer-Verlag, 2012.

[4] J. Appavoo, D. Schatzberg, J. Cadden, and O. Kreiger, “Ebbrt,” *DOE ASCR OS/R Workshop*, 2012.

[5] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, “Work stealing and persistence-based load balancers for iterative overdecomposed applications,” in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pp. 137–148, ACM, 2012.

[6] J. Daily, S. Krishnamoorthy, and A. Kalyanaraman, “Towards scalable optimal sequence homology detection,” in *Workshop on Parallel Algorithms and Software for Analysis of Massive Graphs (ParGraph)*, 2012.

[7] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, “Steal tree: Low-overhead tracing of work stealing schedulers,” in *PLDI*, ACM, 2013.

[8] H. Arafat, J. Dinan, S. Krishnamoorthy, P. Balaji, , and P. Sadayappan, “Work stealing for GPU-accelerated parallel programs in a global address space framework,” *Concurrency and Computation: Practice and Experience special issue on ”Productive Programming Models for Exascale” (under submission)*, 2013.

[9] W. Ma and S. Krishnamoorthy, “Data-driven fault tolerance for work stealing computations,” in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 79–90, ACM, 2012.