RC25555 (WAT1509-050) September 14 7, 2015
Computer Science

IBM Research Report

Exploring the APGAS Programming Model Using the LULESH Proxy Application

Josh Milthorpe, David Grove, Benjamin Herta, Olivier Tardieu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

Research Division
Almaden – Austin – Beijing – Brazil – Cambridge – Dublin – Haifa – India – Kenya – Melbourne – T.J. Watson – Tokyo – Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Many reports are available at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Exploring the APGAS Programming Model using the LULESH Proxy Application

Josh Milthorpe, David Grove, Benjamin Herta, Olivier Tardieu

IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
{jjmiltho,groved,bherta,tardieu}@us.ibm.com

Abstract

The Asynchronous Partitioned Global Address Space (APGAS) programming model enables programmers to express the parallelism and locality necessary for high performance scientific applications on extreme-scale systems. We used the well-known LULESH hydrodynamics proxy application to explore the performance and programmability of the APGAS model as expressed in the X10 programming language. By extending previous work on efficient exchange of ghost regions in stencil codes, and by taking advantage of enhanced runtime support for collective communications, the X10 implementation of LULESH exhibits superior performance to a reference MPI code when scaling to many nodes of a large HPC system. Runtime support for local parallel iteration allows the efficient use of all cores within a node. The X10 implementation was around 10% faster than the reference version using C++/OpenMP/MPI, across a range of 125 to 4,096 places (750 to 24,576 cores). Our improvements to the X10 runtime have broad applicability to other scientific application codes.

Keywords parallel programming models, Partitioned Global Address Space, asynchronous tasks, proxy application

1. Introduction

LULESH [3] is a proxy app for hydrodynamics on an unstructured mesh. It models the Sedov problem: an expanding shock wave in a single material originating from a point blast. The simulation iterates over a series of time steps up to a chosen end time. At each time step, node-centered kinematic variables and element-centered thermodynamic variables are advanced to a new state. The new values for each node or element depend on the values for neighboring nodes and elements at the previous time step. Version 2 of LULESH adds multiple regions and variable cost functions, to more closely simulate the irregular workload of a full hydrodynamics code [5].

LULESH provides opportunities for exploiting parallelism at multiple levels, including vectorization, threading and distributed parallelism. A reference implementation is provided using C++, OpenMP and MPI. We ported the reference implementation of LULESH 2.0 to X10, with some modifications. Whereas the MPI version includes direction- and domain-specific code for communication of ghost data between neighboring processes, the X10 ghost region updates are general-purpose and therefore require far fewer lines of code. The X10 code makes heavy use of function literals (also called lambdas) to improve readability and avoid repetition. As a result, the X10 code is significantly more concise: 2,730 source lines of code for X10 vs. 4,750 for the reference code. Our full source code is available freely in the X10 applications repository [1].

To achieve high performance for LULESH at scale, it was necessary to improve some aspects of the X10 runtime. By implementing X10 collective communication on top of MPI-3 non-blocking collectives [10], we combine the performance of an optimized MPI implementation with the flexibility of active messages in the APGAS model. We also used LULESH to study the performance and implementation alternatives for the `foreach` statement, an efficient mechanism for parallel iteration in X10. This work represents the first large-scale performance evaluation of an X10 application that exploits both single-node (multithreaded) and multi-node (distributed) parallelism.

We begin by surveying related work in Section 2 and reviewing the APGAS programming model in Section 3. Next, we discuss the major technical enhancements to the X10 system and the X10 implementation of LULESH; Section 4.1 covers ghost region exchange, Section 4.2 outlines the integration of MPI-3 non-blocking collectives, and Section 4.3 describes support for parallel iteration. Section 5 reports on overall application performance and finally Section 6 concludes.

2. Related Work

LULESH is a well-studied application code which has been ported to several different programming models. Karlin et al. [4] use the LULESH proxy application to evaluate eight traditional and emerging parallel programming models. Their evaluation includes the Chapel language, which like X10 combines the PGAS programming model with dynamic tasking. However, the Chapel implementation of LULESH focuses on programmability rather than performance and does not exploit multi-node parallelism.

Zheng et al. [17] present UPC++, a PGAS extension to C++ that includes asynchronous remote function invocation. They evaluate the programmability and performance of UPC++ using five example codes. Porting LULESH to UPC++ requires little programming effort and improves performance relative to the MPI reference code, however, the UPC++ port does not exploit multithreaded parallelism within a process. Kumar et al. [6] extend UPC++ with work-stealing within a node. For LULESH, they observe a speedup of 3.4x when using all 12 cores on their test nodes, compared to using only a single core.

Murata et al. [11] describe techniques for obtaining high performance in porting C/MPI applications to X10, using the CoMD and MCCK proxy applications. They discuss optimizations for memory allocation and object access, many of which were used in the X10 port of LULESH. Limpanuparb et al. [7] evaluate the performance of a quantum chemistry code in X10 which uses both multi-threaded and distributed parallelism. However, they focus on a fixed problem size, and therefore do not observe scaling above 512 cores.

Milthorpe and Rendell [9] previously presented an implementation of ghost region updates for X10 distributed arrays. We improve on this implementation by taking advantage of support for bulk one-sided transfers [16], which eliminates allocation and copying costs associated with standard X10 messages.

3. APGAS Programming Model

The X10 programming language [2, 12] has been developed as a simple, clean, but powerful and practical programming language for scale out computation. Its underlying programming model, the APGAS (Asynchronous Partitioned Global Address Space) model [13], is organized around the two notions of *places* and *asynchrony*. A place is an abstraction of shared, mutable data and worker threads operating on the data, typically realized as an operating system process. A single APGAS computation may consist of hundreds or potentially tens of thousands of places.

Asynchrony is provided through a single block-structured control construct, **async** S. If S is a statement, then **async** S is a statement that executes S in a separate thread of control (*activity* or *task*). Dually, **finish** S executes S, and waits for all tasks spawned (recursively) during the execution of S to terminate, before continuing. Memory locations in one place can contain references (*global refs*) to locations at other places. To compute upon data at another place, the **at** (p) S statement must be used. It permits the current task to change its place of execution to p, execute S at p and return, leaving behind tasks that may have been spawned during the execution of S. The termination of these tasks is detected by the **finish** within which the **at** statement is executing. The values of variables used in S but defined outside S are serialized, transmitted to p, and de-serialized to reconstruct a binding environment in which S is executed. Constructs are provided for unconditional (**atomic** S) and conditional (**when** (c) S) atomic execution.

More information on X10 and APGAS can be found online at http://x10-lang.org including the X10 language specification [14], programmer's guide [15], and a collection of tutorials and sample programs.

4. Implementing LULESH in X10

To implement LULESH in X10 so as to exploit both multi-threaded and distributed parallelism, the primary challenges were: i) efficient exchange of ghost region data between neighboring places; ii) collective reduction of time constraints across all places; and iii) local parallel iteration within the numerical kernels of the application. In the following subsections, we consider each in turn.

4.1 Ghost Region Updates

In the reference code, the LULESH simulation domain is divided between MPI processes; similarly, in our code, the domain is divided between X10 places. Each place holds a rectangular block of elements, as well as the nodes that define those elements. A series of stencil operations are applied to update the element and node quantities at each time step. As the stencil operations require element- or node-centered values from a local neighborhood, it is necessary to exchange boundary or ghost regions between neighboring processes. Each place exchanges ghost regions across plane, edge and corner boundaries (in the case of kinematic variables) or plane boundaries only (in the case of artificial viscosity gradients).

In the X10 code, ghost region exchange is encapsulated within the GhostManager class. The design extends a previous implementation using X10 active messages [9]. During program initialization, each place instantiates four GhostManager instances to communicate nodal mass, position and velocities, force, and gradient information. Based on the position of its place in the simulation grid and the kind of information being exchanged, each GhostManager instance computes the set of neighbors with which it will exchange data and pre-allocates the necessary storage to send/receive the data.

Figure 1 illustrates the usage and internal implementation of the GhostManager. In the main application code, an entire ghost region exchange is encapsulated within two function calls (lines 2 and 3). The rest of Figure 1 shows the X10 code necessary to implement this loosely synchronized exchange of data. In lines 8-13, the sending place gathers the needed simulation data into buffer data. The **at** statement at line 14 spawns a remote task at the receiving place, which at line 17 calls the Rail.copy[1] method to transfer the con-

[1] The Rail class represents a one-dimensional, zero-based, densely-indexed array, analogous to a C-style array.

```
1  // Invocation of GhostManager API by LULSEH application
2  forceGhostMgr.gatherBoundariesToCombine();
3  forceGhostMgr.waitAndCombineBoundaries();
4
5  // Send boundary data from this place to neighboring places to be
6  // combined later by waitAndCombineBoundaries.
7  public def gatherBoundariesToCombine() {
8    val src_ls = localState();
9    val sourceDom = src_ls.domainPlh();
10   val sourceId = here.id;
11   for (i in src_ls.neighborListSend.range) {
12     val data = src_ls.sendBuffers(i);
13     sourceDom.gatherData(data, src_ls.sendRegions(i), src_ls.accessFields, src_ls.sideLength);
14     at(Place(src_ls.neighborListSend(i))) @Uncounted async {
15       val dst_ls = localState();
16       val sender = dst_ls.getNeighborNumber(sourceId);
17       Rail.copy(data, 0, dst_ls.recvBuffers(sender), 0, data.size);
18       atomic dst_ls.neighborsReceivedCount++;
19     }
20   }
21 }
22
23 // Wait for all boundary data to be received from neighboring places,
24 // and then combine it with boundary data computed at this place.
25 public final def waitAndCombineBoundaries() {
26   val ls = localState();
27   when (ls.neighborsReceivedCount == ls.neighbotListRecv.size) {
28     val dom = ls.domainPlh();
29     for (i in ls.recvBuffers.range) {
30       dom.accumulateBoundaryData(ls.recvBuffers(i), ls.recvRegions(i), ls.accessFields, ls.sideLength);
31     }
32     ls.neighborsReceivedCount = 0;
33   }
34 }
```

Figure 1: Code snippets illustrating the exchange of ghost regions using the `GhostManager`. The top snippet shows the application invocation of the send and receive operations of the `GhostManager`. The middle snippet uses the **`at`**, **`async`**, and **`atomic`** constructs to asynchronously send ghost data to the neighboring places. The bottom snippet uses **`when`** to wait for ghost data to be received from all neighboring places.

tents of `data` (which was automatically captured and copied by the implementation of **`at`** to the destination place) into a pre-allocated receive buffer at that place. Line 18 atomically increments a count of ghost regions received by the destination place. After sending all of its data, a place invokes `waitAndCombineBoundaries`. At line 27, the **`when`** statement suspends the executing task until it has received updates from all its neighbors (the dual of the increment at line 18). The remainder of the method scatters the received data into the simulation data structures encapsulated in the `Domain` class.

Although succinct and idiomatic, the straightforward **`at`**-based communication does result in an extra memory allocation and corresponding data copy operation in both the sending and receiving places. This happens because the implementation of **`at`** serializes all of the values captured by the **`at`**'s body (`sourceId`, `data`) into a message buffer at the sending place and deserializes them into the program heap at the receiving place. Figure 2 shows how this overhead can be eliminated by using an advanced aspect of the `x10.lang.Rail` API: support for direct one-sided bulk data transfer from one place to another via the `asyncCopy` and `uncountedCopy` methods. Because the ghost regions being exchanged are fairly large, this purely local optimization eliminates several gigabytes of temporary memory allocation and copying overhead per place and yields an 11% overall improvement in LULESH performance.

On network transports that support RDMA, the X10 runtime system internally implements the `asyncCopy` and `uncountedCopy` operations of `Rail` using RDMA. We experimented with implementing these operations using MPI-3 RMA operations. However, we found that on the Cray XE6 system we were using, actually using RMA operations to implement the 'put' variant of `uncountedCopy` that is heavily

```
public def gatherBoundariesToCombine() {
    val src_ls = localState();
    val sourceDom = src_ls.domainPlh();
    val sourceId = here.id;
    for (i in src_ls.neighborListSend.range) {
        val data = src_ls.sendBuffers(i);
        sourceDom.gatherData(data, src_ls.sendRegions(i), src_ls.accessFields, src_ls.sideLength);
        Rail.uncountedCopy(data, 0, src_ls.remoteRecvBuffers(i), 0, data.size, ()=> {
            val dst_ls = localState();
            atomic dst_ls.neighborsReceivedCount++;
        });
    }
}
```

Figure 2: Optimization of ghost data exchange by replacing the `at(...)` `@Uncounted` `async` statement shown in lines 14-19 of Figure 1 by a call to the one-sided bulk transfer method `Rail.uncountedCopy`. The optimization is purely local; no source code changes outside of this code snippet are required.

used in our LULESH code degraded performance by 15% vs. our original implementation that emulated an RDMA put using vanilla ISend/IRecv operations. In future work, we would like to further explore this result on a wider range of MPI-3 implementations.

4.2 Enhanced Collective Operations

The X10 runtime is designed to be deployed on a wide range of distributed systems and adapts to a number of network transports including TCP/IP, versions of MPI, and IBM's PAMI transport.

HPC systems and transports provide advanced communications primitives such as collective operations that are critical to scalable performance. Collectives are made available in X10 through the `x10.util.Team` API which supports operations such as Barrier, Broadcast, and AllReduce over groups of places (`x10.lang.PlaceGroup`). The X10 runtime maps these collectives to the corresponding transport APIs when available or emulates them using point-to-point messages.

The MPI adapter for X10 was originally developed for MPI-2. While MPI collectives perform and scale better than our emulation, this adapter could not fully support the APGAS programming model in X10 because MPI-2 collectives are blocking calls. Concurrent tasks in the same X10 place must be able to interact independently with the network and make progress irrespective of pending collective operations in other tasks. In practice, programmers had to funnel all communications through one master task in each place to make use of this transport.

To better support X10 on MPI, we have introduced a new MPI-3 adapter. By using MPI-3 non-blocking collectives for Barrier, AllReduce and so on, we can directly support application codes such as LULESH which naturally overlap remote task invocations (ghost region exchanges) and collective operations (reductions).

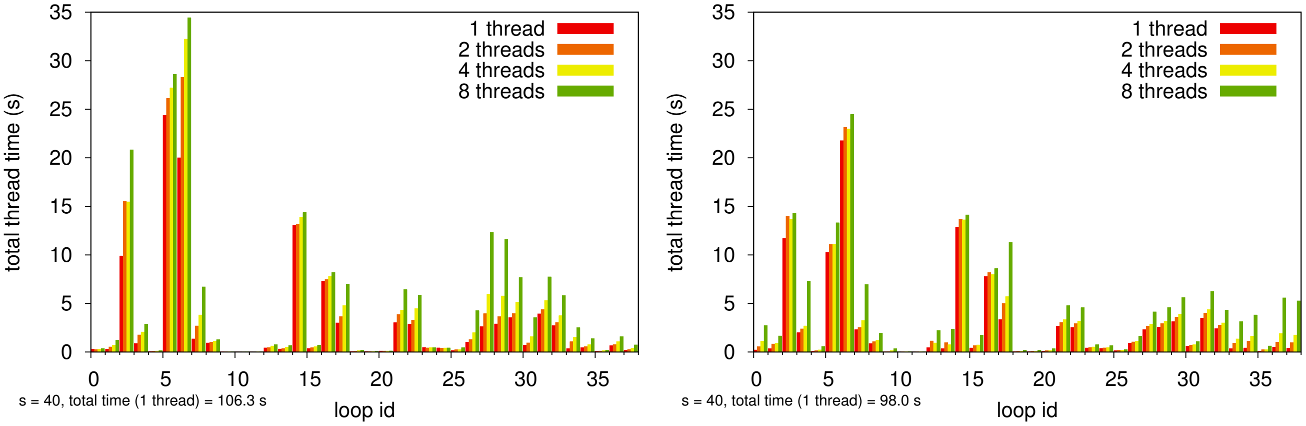
4.3 Scheduling Local Parallel Iterations

The reference version of LULESH 2 exploits multithreaded parallelism using OpenMP. A total of 38 loops are parallelized using the parallel worksharing directive `#pragma omp parallel for`. Several of these loops include the `nowait` specifier, meaning that a compiler may choose to fuse these loops with subsequent loops. In the X10 version of the code, we chose to fuse these loops, as well as others (numbered 18–20 and 21–31 and 33–34). (These fusions could also be applied to the OpenMP reference code.)

In a previous paper, Milthorpe [8] presented the **foreach** statement, an efficient mechanism for parallel iteration in X10. We used **foreach** to parallelize 34 out of the 38 loops. The remaining loops (numbered 0 and 9–11 in figure 3) were too small to profit from parallel execution with **foreach**.

The **foreach** library implements various strategies for dividing work along an iteration space into multiple independent tasks that can be scheduled in parallel by the X10 runtime. It relies on X10's **finish** construct to detect the termination of the loop. The X10 runtime uses work stealing to schedule the tasks, which is efficient—low overhead—and effective—high utilization—but does not permit controlling task affinity, e.g., mapping the same iteration subspaces to the same cores across loop nests. To investigate, we implemented a separate scheduler just for **foreach**. It provides a fixed-sized thread pool, thread binding, deterministic mapping from iteration subspaces to threads, and lightweight synchronization barriers. While in theory, this scheduler could perform a little better than X10's work-stealing scheduler at the expense of generality, we did not observe a significant performance advantage. All the experimental results in this paper therefore rely on X10's default scheduler.

We ran both the C++/OpenMP and the X10 versions of LULESH for a 40^3 mesh on an Intel Ivy Bridge-EP Xeon E5-4657L v2@2.4 GHz. The machine has four sockets, each



(a) C++ / OpenMP

(b) X10

Figure 3: LULESH parallel loop scaling on Ivy Bridge-EP: X10 vs. C++/OpenMP.

with 12 cores. Figures 3(a) and 3(b) show scaling with number of threads for the C++/OpenMP and X10 codes respectively, for each of the 38 loops. ‘Total thread time’ is the sum of thread times across all threads; in this plot, perfect parallel efficiency would result in a constant total thread time for increasing number of threads.

Overall, the X10 code was faster for a single thread (98.0 s compared to 106.3 s), and exhibited superior speedup with increasing number of threads. In particular, the major loops (2: *IntegrateStressForElems*, 5–6: *CalcHourglassControlForElems* and 14: *CalcKinematicsForElems*) all exhibit near-constant total thread time in X10. In contrast, the total thread time for loops 2, 5 and 6 all increase rapidly with number of threads in the C++/OpenMP version, indicating a loss of parallel efficiency. Loops 37 and 38 exhibit significantly worse scaling for X10 than for C++/OpenMP, due to poor locality between these loops and the earlier loops (14,15,21) that write the data they require. However, the total runtime of loops 37 and 38 is small compared to other loops in the application.

5. Performance At Scale

To evaluate the overall performance of the X10 LULESH code, we utilized a Cray XE6 system at NERSC called Hopper. Each Hopper compute node consists of two twelve-core AMD ‘MagnyCours’ 2.1 GHz processors and 32 GB of memory. The 24 cores in a compute node are grouped into 4 NUMA domains with a shared 6 MB L3 cache. For all of our experiments, we scheduled one place (MPI rank) per NUMA domain. Thus each place had 6 cores available for use by the multithreaded parallelism constructs discussed in the previous section. We performed weak-scaling experiments; each place was assigned 64,000 elements (40^3). The primary metric is ‘Grind Time’, a measurement of the per-element compute time that is reported by the LULESH application. Lower Grind Times represent better performance (less time taken per unit of work). All data points represent the median of 5 runs; run-to-run variation at the same configuration was less than 2%.

Figure 4 shows the overall performance of the X10 and C++/OpenMP/MPI implementations of LULESH as we scaled the number of cores from 6 (1 place) to 24,576 (4,096 places). At all problem sizes, X10 achieved better performance than the C++/OpenMP/MPI reference version of the code. Single-place performance was 26% better. As the experiments reached more realistic scales, the performance gap was approximately 10% (with an exception at 1,728 places where the gap was only 4% it otherwise ranged from 8% to 12% over all measurements between 125 to 4,096 places).

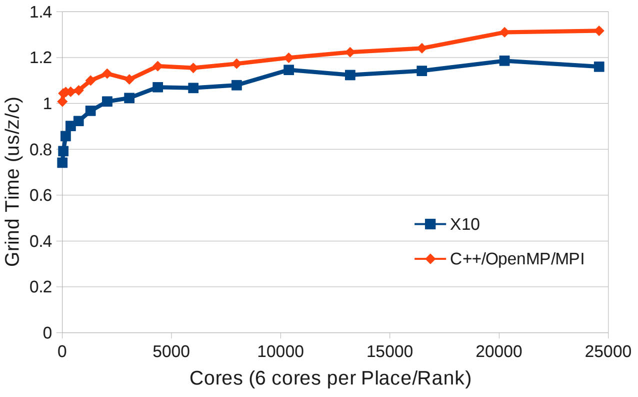


Figure 4: LULESH performance on Hopper (Cray XE6) from 6–24,576 cores (1–4,096 places). Lower ‘Grind Time’ is better.

Table 1 reports on the parallel speedup within a single place by comparing single-threaded performance (where X10 and OpenMP are restricted to using a single worker thread per place) to that of the peak configuration (five worker threads per 6 core place). Compared to their own single-threaded versions, multithreaded X10 achieved a 3.77 times speedup and multithreaded OpenMP achieved a 3.31 times speedup. The speedup shown in Table 1 was measured for runs using 27 places.

	Grind time (μs/z/c)		**Speedup**
	1 thread	5 threads	
X10	**3.28**	**0.87**	**3.77×**
C++/OpenMP/MPI	3.47	1.05	3.31×

Table 1: LULESH multithreaded speedup on Hopper.

6. Conclusion

The X10 implementation of LULESH represents the first large-scale performance evaluation of an X10 application exploiting both single-node (multithreaded) and multi-node (distributed) parallelism. The X10 implementation was roughly 10% faster than the reference C++/OpenMP/MPI implementation and exhibited good scaling up to 4,096 places (24,576 cores). The X10 code was also more concise, with only 2,730 lines of code compared to 4,750 lines for the reference implementation.

Our improvements to collective support in the X10 runtime have broad applicability to other scientific codes.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Award Number DE-SC0008923.

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

All source code line measurements were generated using David A. Wheeler's 'SLOCCount'.

References

[1] X10 Applications Repository. Tag: RESPA2015. `https://github.com/x10-lang/x10-applications/releases/tag/RESPA2015`.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*, pages 519–538, 2005.

[3] R. D. Hornung, J. A. Keasler, and M. B. Gokhale. Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254, Lawrence Livermore National Laboratory, 2011.

[4] I. Karlin, A. Bhatele, J. Keasler, B. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring traditional and emerging parallel programming models using a proxy application. In *Proc. 27th International Parallel and Distributed Processing Symposium (IPDPS 2013)*. IEEE, 2013.

[5] I. Karlin, J. Keasler, and R. Neely. LULESH 2.0 updates and changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, August 2013.

[6] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar. HabaneroUPC++: A compiler-free PGAS library. In *8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014.

[7] T. Limpanuparb, J. Milthorpe, and A. Rendell. Resolutions of the Coulomb operator: VIII. Parallel implementation using the modern programming language X10. *Journal of Computational Chemistry*, 35(28):2056–2069, Oct 2014.

[8] J. Milthorpe. Local parallel iteration in X10. In *Proc. ACM SIGPLAN Workshop on X10*, pages 7–12. ACM, 2015.

[9] J. Milthorpe and A. Rendell. Efficient update of ghost regions using active messages. In *Proc. 19th IEEE International Conference on High Performance Computing (HiPC 2012)*. IEEE, Dec 2012.

[10] MPI Forum. MPI: A message-passing interface standard version 3.0. Technical report, MPI Forum, Sep 2012. URL `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`.

[11] H. Murata, M. Horie, K. Shirahata, J. Doi, H. Tai, M. Takeuchi, and K. Kawachiya. Porting MPI based HPC applications to X10. In *Proc. ACM SIGPLAN Workshop on X10*. ACM, 2014.

[12] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Concur'05*, pages 353–367, 2005.

[13] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The Asynchronous Partitioned Global Address Space Model. In *Proc. First Workshop on Advances in Message Passing (AMP'10)*, June 2010.

[14] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. The X10 language specification, v2.5. June 2015.

[15] V. Saraswat, O. Tardieu, D. Grove, D. Cunningham, M. Takeuchi, and B. Herta. A brief introduction to X10 (for the high performance programmer). `http://x10.sourceforge.net/documentation/intro/latest/html/`, June 2015.

[16] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 and APGAS at petascale. In *Proc. 19th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '14)*, pages 53–66. ACM, 2014.

[17] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: a PGAS extension for C++. In *Proc. 28th International Parallel and Distributed Processing Symposium (IPDPS 2014)*, pages 1105–1114. IEEE, 2014.