X10 as a parallel language for scientific computation: practice and experience

Josh Milthorpe
V. Ganesh
Alistair P. Rendell
School of Computer Science
Australian National University
{*josh.milthorpe,ganesh.venkateshwara,alistair.rendell*}*@anu.edu.au*

David Grove
Thomas J. Watson Research Center
IBM
groved@us.ibm.com

***Abstract*—X10 is an emerging Partitioned Global Address Space (PGAS) language intended to increase significantly the productivity of developing scalable HPC applications. The language has now matured to a point where it is meaningful to consider writing large scale scientific application codes in X10. This paper reports our experiences writing three codes from the chemistry/material science domain: Fast Multipole Method (FMM), Particle Mesh Ewald (PME) and Hartree-Fock (HF), entirely in X10. Performance results are presented for up to 256 places on a Blue Gene/P system.**

During the course of this work our experiences have been shared with the X10 development team, so that application requirements could inform language design discussions as the language capabilities influenced algorithm design. This resulted in improvements in the language implementation and standard class libraries, including the design of the array API and support for complex math.

Data constructs in X10 such as *places* and *distributed arrays*, and parallel constructs such as *finish* and *async*, simplify implementation of the applications in comparison with MPI. However, current implementation limitations in X10 2.1.2 make it difficult to achieve scalable performance using the most natural expressions of the algorithms. The most serious limitation is the use of point-to-point communication patterns, rather than collectives, to implement parallel constructs and array operations. This issue will be addressed in future releases of X10.

***Keywords*-X10; Partitioned Global Address Space (PGAS); parallel languages; scalability; scientific computing; Fast Multipole method; Smooth Particle Mesh Ewald method; Hartree-Fock method**

I. Introduction

Identifying and exploiting parallelism is the key to effectively harnessing massive computing power for real-world applications. Over the last decade or so the shared-memory programming model has supported the productive development of high performance codes. However, as we enter the many-core era the increasing relative cost of cache coherency is shifting the balance towards distributed memory models. The dominant distributed memory programming model is message passing, exemplified by MPI. It has been used successfully to develop high-performance codes scaling to hundreds of thousands of cores. However, message-passing is much less productive than the shared-memory model (for example, OpenMP)[1].

In 2002, with the aim of reducing both development and execution time of HPC applications, DARPA initiated the High Productivity Computing Systems (HPCS) program under which the X10 programming language is being developed[2]. X10 is a Partitioned Global Address Space (PGAS) language, which explicitly represents locality in the form of places. A *place* in X10 corresponds to a processing element with attached local storage, with each place supporting a set of dynamically spawned lightweight *activities*. Activities specify logical parallelism in the form of structured and unstructured constructs (*ateach*, *async* and *Future*). An activity runs to completion at the place where it was started, but may spawn new remote activities (*at*), and detect termination of spawned activities (*finish*). Activities coordinate using *clocks* and lock-free synchronization (*atomic*). X10 also provides a rich array sublanguage. An *array* is a collection of objects which are indexed by *points* in a bounded *region*. Each element in a distributed array is assigned to a particular place according to the array *distribution*, which is a mapping from point to place[3]. The X10 language specification, programmer's guide and tutorials are available at http://x10-lang.org.

Over the life of the HPCS program, the X10 language has evolved through a number of iterations. With the program now drawing to a close, X10 2.1 brings a number of changes aimed at interoperability with existing Java and Scala codes, which are expected to be the last major changes before broader release. Compiler and tool support have also greatly improved. It is therefore timely to review the progress of X10 towards widespread applicability.

This paper presents initial implementations and our experience in writing three different scientific codes entirely in X10 using version 2.1.2 of the language specification and implementation. The first two of these codes calculate the electrostatic potential using the Fast Multipole method (FMM) and the Smooth Particle Mesh Ewald (PME) method. The third performs a complete Hartree-Fock (HF) quantum chemistry computation on a molecular system.

Section II of this paper gives a brief background of the applications considered. Section III illustrates how data distribution constructs in X10 such as *places* and distributed *arrays*, and parallel constructs such as *finish* and *async*, greatly simplified the parallel implementation of these applications.

Section IV presents a preliminary performance analysis of the code. We report some implementation limitations of X10 2.1.2, which made it difficult to achieve high performance using the most natural expression of the algorithms. While these limitations should be removed in future versions of the language, where possible we suggest techniques for working around them in the interim.

II. Background of applications

We selected three applications, which are representative of different patterns of computation and communication. Following Colella's 'Seven Dwarfs'[4], the applications represent N-body methods (FMM and HF), spectral methods (PME) and dense linear algebra (used in the HF code).

A. Fast Multipole Method (FMM)

Fast Multipole methods are used in diverse fields to efficiently evaluate long-range interactions in N-body problems, reducing the computational complexity from $O(N^2)$ to $O(N)$. Our application code uses the Fast Multipole method to calculate long-range electrostatic potential between particles in a molecular mechanics simulation. In the 3D FMM[5], the simulation space is divided into an octree of cubic boxes. Interactions between particles in nearby boxes are evaluated directly, whereas distant interactions are evaluated by means of analytic expansions around box centers or equivalent densities[6]. The parallel FMM is characterized by complex data structures and localized communication patterns. The algorithm comprises several steps:

- generation of far-field representations (*P2M*);
- an upward pass combining representations at higher levels in the tree (*M2M*);
- transformation to local representations (*M2L*);
- a downward pass translating local representations to lower levels (*L2L*);
- evaluation of far-field interactions using local representations (*L2P*); and
- direct evaluation of near-field interactions (*P2P*).

Our code uses the expansions and operations described by White and Head-Gordon[7] and rotation matrix relations by Dachsel[8] with extensions for periodic boundary conditions due to Lambert, Darden and Board[9] and Kudin and Scuseria[10].

B. Particle Mesh Ewald (PME)

Like the Fast Multipole method, the Smooth Particle Mesh Ewald method[11] is used in molecular dynamics simulations to evaluate long-range electrostatic interactions. Unlike the FMM it is limited to systems with periodic boundary conditions. The interaction is split into short-range and long-range components. The algorithm comprises several steps:

- a *charge density field* Q is approximated by interpolating charges on a mesh of grid points;
- the inverse FFT $F^{-1}(Q)$ of the charge array is calculated;
- $F^{-1}(Q)$ is multiplied in reciprocal space with a *pair potential* array $F^{-1}(\Theta_{rec})$;
- the result is transformed by a forward FFT to give the convolution of $\Theta_{rec} \star Q$;
- the reciprocal potential is calculated as the entrywise product $(\Theta_{rec} \star Q) \circ Q$
- the short-range component is calculated directly within a reduced domain; and
- a correction is applied to cancel the self-interaction of each particle.

Forces and potentials for each particle are also interpolated between the grid points. The long-range potential is calculated in reciprocal space, which places two 3D Fast Fourier Transforms (FFTs) (inverse and forward) at the core of this method. A key feature of the distributed 3D FFT is the need for all-to-all communication for the 3D transpose[4]. The computational scaling of this method is $O(N \log N)$.

C. Hartree-Fock (HF)

The Hartree-Fock method is widely used in quantum chemistry[12] and involves solving the following pseudo-eigenvalue problem:

$$FC = \epsilon SC \tag{1}$$

Where F, C and S are the Fock, molecular orbital Coefficient and Overlap matrices respectively, the dimensions of which are dependent on the number of basis functions used to represent the molecular system, and therefore indirectly on the number of atoms in the system. We solve for C, with a known constant matrix S. However, F is dependent on C, which necessitates use of an iterative Self Consistent Field (SCF) procedure to solve these equations. The most expensive part in the above equation is setting up of the Fock matrix (F):

$$F_{ij} = H_{ij}^{core} + \sum_{k,l}^{N} P_{kl}[\langle ij|kl\rangle - \frac{1}{2}\langle ik|jl\rangle] \tag{2}$$

where i, j, k, l denote atom centered basis function indices, $\langle ij|kl\rangle$ is a four centered two-electron integral and H^{core} is a matrix containing one-electron integrals. P is the density matrix and is obtained from C.

In principle, formation of the Fock matrix is an $O(N^4)$ operation, for a molecular system represented using N basis functions. Our implementation uses the scheme described by McMurchie and Davidson [13] for evaluating the two-electron integral terms $\langle ij|kl\rangle$. We have implemented a direct SCF method, that re-evaluates these integrals for each iteration of the SCF[14]. The individual two-electron integrals can be computed independently, but incur vastly different computational costs. Further, each two-electron integral contributes to six different locations in the Fock matrix[14]. Overall, this problem is characterized by load imbalance in the evaluation of the two-electron integrals and the need to add multiple contributions to every element in the F matrix.

The SCF procedure also requires a matrix diagonalization of size N, and other linear algebra operations. For moderately sized systems, the cost of these operations is negligible compared to the cost of computing the F matrix.

III. Implementation perspective

This section highlights some key issues in implementing the above applications in X10. We first look at some of the general requirements for writing all the applications. We then discuss each of the application codes in turn and the features of X10 that were helpful in writing these applications. Finally we highlight some caveats and current implementation limitations of X10 which were encountered while developing these codes.

A. Basic requirements of applications

A typical scientific code requires both language support for numerical computation and optimized numerical libraries. For the FMM and PME codes, one major requirement is efficient support for complex math and arrays of complex numbers. When we began this work, X10 did not provide a standard `Complex` type. The combination of X10 *structs* and user-defined operators enabled us to add complex number support to X10 in a natural and efficient way. A *struct* in X10 is essentially a restricted object that is implemented inline, without any object header overhead or other indirection. Structs are designed specifically to enable the creation of space-efficient user-defined compound types. The code snippets below show part of the definition of the `Complex` type and some usage examples.

```
public struct Complex {
  public val re:Double;
  public val im:Double;
  public operator this +
    (that:Complex):Complex {
      return Complex(re + that.re,
                     im + that.im);
  }
  public operator this *
    (that:Complex):Complex {
      return Complex(re * that.re
                   - im * that.im,
                     re * that.im
                   + im * that.re);
  }
  ...
}
```

```
val x:Complex = Complex(3,4);
val y:Complex = ...;
val z:Complex = x*y + 2*x;
```

Our implementation of the *Complex* type was incorporated in the X10 standard libraries and has been available to all X10 users since X10 2.0.0.

X10 supports full interoperability with C++ and Java through the `@Native` annotation on classes, methods or blocks. For the HF code, implementing the SCF procedure required linear algebra operations. Where appropriate these were provided by native calls to the GNU Scientific Library[15]. The PME code uses FFTW[16] for all fast Fourier transforms.

The object-oriented nature of X10 made it possible to build extensible class libraries that were used for all the applications. For instance, class definitions like `Atom`, `Molecule` and `Vector3d` are shared for all the three applications. For the HF code all linear algebra operations were written in separate shared classes, making it easy to reuse these classes in other codes as well as optimize these at a later stage without changing the application code. Each type of matrix in HF code extends a top level `Matrix` class allowing for easy type identification and resulting in a more readable code. Furthermore, the X10 standard libraries are designed to allow applications to extend and customize their functionality; this was particularly useful for the array regions and distributions used in the FMM code.

B. FMM

A key requirement for the parallel Fast Multipole Method is the construction and manipulation of a distributed octree of boxes. Each place maintains its own local portion of the tree, and information about the overall geometric partitioning of the tree[17]. This is accomplished in X10 through the use of distributed arrays. The following code creates a distributed array for a single level of the octree:

```
val dim = Math.pow2(levelNum) as Int;
val thisLevelRegion : Region(3) =
  0..(dim-1) * 0..(dim-1) * 0..(dim-1);
val thisLevelDist =
  MortonDist.make(thisLevelRegion);
thisLevel = DistArray.make[FmmBox]
              (thisLevelDist);
```

While the standard X10 class libraries provide a number of commonly used distributions including *block*, *cyclic* and *unique*, the user may create custom subclasses of `x10.array.Dist` to support more complex geometric decomposition. The `MortonDist` class in the listing above is an example of such a distribution that divides a three-dimensional array between places using Morton- or Z-indexing, which is the most efficient distribution for the FMM octree[18].

Arrays are not limited to dense, rectangular regions. For example, multipole expansions O_{lm} have a peculiarly shaped region $0 < m \leq p$, $l \leq |m|$. This is represented in our code by a custom subclass of `x10.array.Region` named `ExpansionRegion`.

```
val expRegion = new ExpansionRegion(numTerms);
val expTerms = new Array[Complex](expRegion);
for ([i,j] in expTerms) {
    expTerms(i,j) = ...
}
```

The tight integration of the Array API into the language allows a clean separation of data layout and distribution from other concerns.

X10 provides a special struct type `GlobalRef`, which is a global reference to an object at one place that may be passed to other places. `GlobalRef` can be used to build distributed data structures that are not best represented using distributed arrays. In the FMM code, it is used in each box of the octree to store a pointer to its parent box, which may be held at a different place. The following code retrieves the local expansion for the parent box, used in the *L2L* translation of local representations downwards through the tree:

```
// at parent place
val parent = GlobalRef[FmmBox](parentBox);
// at child place
val parentExpansion =
    at (parent) {parent().localExp};
```

In MPI, this data would typically be communicated through a pair of `MPI_Send` and `MPI_Recv`, requiring synchronization of source and destination process. Alternatively, the data could be stored in a memory window and accessed using a one-sided `MPI_Get`. However this places other restrictions on the program, for example that data in the same region may not be concurrently updated by remote and local write operations[19]. Thus the style of one-sided communications supported by X10 (and other asynchronous PGAS languages) is more flexible than that available in MPI.

A powerful data-parallel construct, *ateach*, supports parallel execution over the elements of a distributed array. For example, the following code executes in parallel over all boxes in a level:

```
finish {
  ateach (boxIndex in thisLevel) ...
}
```

For a place that supports multiple threads of execution, for example an SMP node within a Linux cluster, *ateach* is also multithreaded within that place. To achieve the same result using MPI requires a hybrid model, with (for example) OpenMP providing parallelism within an SMP node.

C. PME

At the heart of the PME method are the two FFTs performed on the charge mesh. The PME mesh is divided between places in the x and y dimensions, meaning each place holds a "thick pencil" extending through the entire z dimension. Following Eleftheriou et al.[20], the 3D FFT is decomposed into a series of 1D FFTs performed using FFTW[16], interspersed with transpose operations to redistribute the data so that each place's data is complete in one dimension for each FFT. The transpose is an all-to-all communication, with the following X10 code executed at each place:

```
def transpose(
 source : DistArray[Complex](3),
 target : DistArray[Complex](3)) {
  finish {
    for (p2 in source.dist.places()) {
      val transferRegion : Region(3) = //
      val toTransfer = // ...
      async at (p2) {
        var i : Int = 0;
        for ([x,y,z] in transferRegion) {
            // transpose dimensions
            target(z,x,y) = toTransfer(i++);
        }
      }
    }
  }
```

This demonstrates how the "active message" (*async at*) idiom can be used as a building block for a complex pattern of communication.

In MPI, this transpose would be implemented using `MPI_Alltoall`, which is a more efficient implementation choice for this particular algorithm as it combines data transfer with the necessary synchronization of transpose and FFT phases. To support this pattern of communication, X10 has been enhanced to support some collective operations such as all-to-all within *Teams* of places, similar to MPI communicators. Future versions of X10 will support a greater range of collective operations.

D. Hartree-Fock

The most time-consuming part of the Hartree-Fock method is the generation of the two-electron integrals $\langle ij|kl \rangle$, which are multiplied with the appropriate density (P) matrix elements to form the F matrix. As it is difficult to predict *a priori* which elements are required at each place, the density matrix is replicated to all places. The replication is implemented by copying the array to each place using *ateach*, as part of the "active message" that initiates computation of the partial contribution to the F matrix.

As noted in Section II, the central concern for parallel implementation of HF is load balancing of the two-electron integral evaluations. We explored several alternative approaches to achieve this goal. The first mechanism is a simple program-managed load balancing using *async* at a single place. The simplest implementation uses a global F matrix with updates synchronized using *atomic* blocks. This is one of the techniques suggested by Bernholdt et al.[21].

A slightly modified version uses local copies of F matrices to avoid using atomic blocks.

```
finish {
 [loop over atom centers]
  [loop over basis functions
    on each atom center <i,j,k,l>]
   async computeAndRecord2E(i,j,k,l);
}
```

For simplicity, the above loop structure in subsequent text will simply be abbreviated as *[i,j,k,l loop]*. Though straightforward, for larger system sizes, this code crashes with an out of memory error when using the current implementation of *async* in X10 2.1.2. This is because as the number of atoms increases there is an $O(N^4)$ increase in the number of *async* activities that are generated. Each activity is represented as a closure object with associated environment, requiring some local memory allocation. Hence, in general, use of *async* in its current implementation, for spawning a large number of fine grained activities should be avoided.

A more prudent approach is to spawn only as many activities as the number of places or threads available. A modified version that uses this approach and which also works for multiple places is given below:

```
// initialize and copy data to places
val computeInst = DistArray.make[ComputePlace](
  Dist.makeUnique(),
  (Point) => new ComputePlace(
    N, molecule, basisName)
);
```

The first stage involves copying all the required data to all the available places. A `ComputePlace` object controls the work and all necessary local data for each place.

```
// copy density matrix to each place
finish ateach (p in computeInst) {
  computeInst(p).reset(density);
}
// compute integrals, and store in
// local F matrix
finish {
 [=> i,j,k,l loop]
   var accepted:Boolean = false;
   while(!accepted) {
     for(p in computeInst) {
       accepted = at(computeInst.dist(p)) {
         (computeInst(p).accept(i,j,k,l);
     }
   }
}
// gather and reduce partial
// contributions to F matrix
finish for ([placeId] in computeInst) async {
  val contrib = at (Place.place(placeId)) {
    return computeInst(p).getContribution();
  };
  val sum = (a:Double, b:Double) => (a+b);
  atomic fock.map[Double,Double]
    (fock, contrib, sum);
}
```

The above code only assigns work to places that are free, thereby achieving even load-balancing. However, it involves communication of the relevant $\langle ij|kl\rangle$ indices to remote places, which requires frequent small messages and thus degrades overall scaling.

An alternative is to use purely static load balancing, with a coarse granularity of work at the outermost loop over the shell indices:

```
// copy density matrix to each place
// and compute local Fock contribution
finish for ([placeId] in computeInst) async {
  val contrib = at (Place.place(placeId)) {
    computeInst(p).reset(density);
    computeInst(p).computeShells(nPairs);
    return computeInst(p).getContribution();
  };
  // gather and reduce Fock contribution
  val sum = (a:Double, b:Double) => (a+b);
  atomic fock.map[Double,Double]
    (fock, contrib, sum);
}
```

Yet another mechanism (also suggested by Bernholdt et al.[21]) is to use dynamic load balancing that is implemented via a shared counter:

```
public class SharedCounter {
  private var counter : GlobalRef[AtomicInteger];

  public def this() {
    val a = new AtomicInteger();
    counter = GlobalRef[AtomicInteger](a);
  }

  public def getAndIncrement() {
    return at(counter) {
      counter().getAndIncrement()
    };
  }
}
...
val G = new SharedCounter();
finish for ([placeId] in computeInst) async {
  val contrib = at (Place.place(placeId)) {
    val comp_loc = computeInst(p);
    comp_loc.reset(density);
    var myG:Int = 0;
    var L:Int = 0;
    val F = Future.make[Int](
      () => G.getAndIncrement());
    myG = F.force();
    [=> i,j,k,l loop]
      if (L == myG) {
        val Fn = Future.make[Int](
          () => G.getAndIncrement());
        comp_loc.computeAndRecord2E(
                   i,j,k,l);
        myG = Fn.force();
      }
      L++;
    // end loop [i,j,k,l]
    return computeInst(p).getContribution();
} }
```

In the above code, each of the places iterates over all tasks, when the counter `L` matches the next assigned task to the place, `myG`, the integral block is evaluated or else skipped. Note that this method also involves frequent communication to the first place, that could in turn degrade overall scaling. A more promising load balancing method is distributed work-stealing, as implemented in the X10 *global load balancing framework*[22]. We have not yet tested this framework for the Hartree-Fock code.

E. Caveats and current implementation limitations

During the initial performance profiling of the applications, it was realized that the Array API of X10 2.0 had some design flaws. In particular, the API did not allow the programmer to distinguish statically between arrays that were known to be local (all of their data contained in a single place) or distributed (data contained in multiple places). As a result, the performance of local arrays suffered as the compiler was unable to inline and fully optimize calls to the `Array.set` and `Array.apply` methods. As a result of our experiences, the Array API was modified, and starting in X10 2.0.3 the class library makes a user-visible distinction between local and distributed arrays. This change resulted in an overall performance gain of about 10-20% for all three applications. The performance of `DistArray` access is still relatively slow, as the API has been designed primarily for flexibility of distribution and data layout, rather than fast access.

For applications that require a large set of activities such as the straightforward implementation of HF kernel (section III-D), the current implementation of *async*, which is based on a Fork-Join style, fails with out-of-memory errors as the number of activities increases. Consequently, the HF code is implemented in a slightly different manner to spawn only as many activities as the number of places. There is work underway in the X10 project to implement Cilk-style work stealing within a single place as the underlying mechanism to implement *async*, but that effort is still in early stages and is not yet usable for large-scale applications.

An important issue that limits the scaling achievable with the current implementation of X10 is the use of point-to-point communication patterns. The `Team` API does provide some collective operations which are mapped to the underlying network-supported collectives for Blue Gene and other platforms. However, key language constructs such as distributed *finish*, *ateach* and `DistArray` reductions are implemented using point-to-point communication at the spawning place. This means that these operations have a cost of $O(P)$, where P is the number of places involved in the computation. For large numbers of places, this cost may be a significant proportion of the program execution time. For optimal scaling, these operations should be implemented on top of tree-based collective functions like those used in standard MPI implementations, which scale as $O(\log P)$. This issue is under consideration and will be addressed in future releases of X10.

IV. Evaluating the applications

A. Performance results

The previous section highlighted how each application benefited in terms of ease of expression of various data structures, data distribution and parallelization. In this section, we look at the initial performance characteristics of our applications run using the current X10 implementation. For all the numbers reported here, the C++ backend implementation of X10 2.1.2 was used[23]. The applications were run on varying numbers of places on the Watson 4P Blue Gene/P system, where each place is one core (four places per node).

Table I presents parallel scaling of the evaluation of electrostatic potential using the FMM code. The target system is a box of 80Å side length containing 17,132 SPC water molecules[24] (51,396 atoms) in a molecular dynamics simulation.

Table I
FMM strong scaling. *Wall-clock time for potential evaluation for a uniform distribution of 51K particles.*

places	time (s)	proportion of single node time	speedup
1	71.5	1.00	1.00
2	37.4	0.52	1.91
4	19.8	0.28	3.61
8	9.87	0.14	7.24
16	5.56	0.078	12.9
32	2.86	0.040	25.0
64	1.57	0.022	45.5
128	1.11	0.016	64.4
256	0.973	0.014	73.5

The code exhibits close to linear scaling up to 64 places. However, scaling is reduced for more than 64 places because for this relatively small problem size, the tree is only four levels deep. A significant proportion of the work is done at higher levels in the tree, and there are not enough boxes in the highest levels to assign at least one box to each place. This means that the communication and computation costs are not balanced evenly between places. We hope to address this issue in a future version, using the notion of *shared octants* discussed in Lashuk et al.[17]. This load imbalance is reduced with larger problem sizes, as illustrated in table II.

For 200,000 particles, there are five levels in the tree, so proportionately more work performed at the lower levels and evenly balanced between places.

The base performance of the FMM code is also poor, and little effort has been made to optimize the *M2M*, *M2L* and *L2L* operators. We plan to use a more efficient BLAS formulation[25] of these operators in a future version.

Table III presents parallel scaling of the evaluation of the long-range potential for the same system of 51,396 atoms using the PME code. This does not include the time to

Table II
FMM STRONG SCALING. *Wall-clock time for potential evaluation for a uniform distribution of 200K particles.*

places	time (s)	proportion of single node time	speedup
1	306.1	1.00	1.00
2	155.8	0.51	1.96
4	83.7	0.27	3.66
8	41.1	0.13	7.45
16	20.9	0.068	14.6
32	10.7	0.035	28.6
64	6.06	0.020	50.5
128	4.76	0.016	64.3
256	2.36	0.008	129.7

calculate direct short-range potential, which is dependent on cutoff distance (10Å in our experiments).

Table III
PME STRONG SCALING (MESH ONLY). *Wall-clock time for PME mesh calculation for 51K particles, grid size* 64^3.

places	time (s)	proportion of single node time	speedup
1	13.7	1.00	1.00
2	10.7	0.78	1.28
4	5.69	0.42	2.41
8	2.76	0.20	4.96
16	1.35	0.099	10.1
32	0.76	0.056	17.9
64	0.45	0.033	30.2
128	0.35	0.025	39.7
256	0.37	0.027	37.3

Scaling for the PME code reduces above 64 places due to the overhead of the *finish ateach* idiom as discussed in section III-E. This idiom is used in the code to express fork-join style parallelism. It is likely that scaling could be improved with the current version of X10 by reorganising the application in a SPMD style.

Table IV presents parallel scaling of energy calculations using the HF code for a benzene molecule using the 3-21G basis set (with 66 basis functions). Of all the schemes tried for parallelization of the HF, only the static load balancing scheme with work distribution over the outermost loop runs to completion, and shows positive scaling. As stated in previous section, for the other cases, the code either runs out of memory or incurs an excessive communication cost. Consequently, we present scaling results using only the static load balancing scheme.

The code does not scale above 256 places due to poor load balancing. This is due to the coarse granularity of the work units relative to the problem size (1764 work units in this static load balancing scheme). The base performance is also poor due to unnecessary recalculation of intermediate products in the integral evaluation; this could be avoided using the PRISM algorithm for two-electron integrals[26].

Table IV
HF STRONG SCALING. *Wall-clock time for Fock matrix formation: Benzene 3-21G (12 atoms, 66 basis functions).*

places	time (s)	proportion of single node time	speedup
1	62.9	1.00	1.00
2	38.6	0.61	1.63
4	20.1	0.32	3.13
8	10.3	0.16	6.11
16	5.30	0.084	11.9
32	3.31	0.053	19.0
64	2.12	0.034	29.7
128	1.29	0.021	48.8
256	0.83	0.013	75.8

B. PME comparison with GROMACS

As a comparison with our PME application code, we ran an energy minimization using GROMACS 4.5[27] on varying numbers of Blue Gene cores, for the same system of 17,132 SPC water molecules described above. Table V presents the scaling of the *PME mesh* component of the calculation, which performs a comparable calculation to our PME code. The base performance of GROMACS is almost two orders of magnitude better than our code. This is disappointing, even given that GROMACS is one of the most highly optimized molecular dynamics packages available. We have only recently begun comparison with GROMACS and we are starting to investigate the causes for the poor relative performance of our code. Our code does not implement an identical algorithm to that used in GROMACS. This clouds performance issues that arise from using X10 versus, for example, C and MPI. We plan to develop as near as possible identical implementations of these kernel applications in C/C++ and MPI, to help better understand the performance results achieved using X10.

Table V
GROMACS STRONG SCALING (PME MESH ONLY). *Wall-clock time for PME mesh calculation for 51K particles, grid size* 64^3.

cores	time (s)	proportion of single node time	speedup
1	0.160	1.00	1.00
2	0.094	0.59	1.70
4	0.049	0.31	3.27
8	0.028	0.18	5.71
16	0.023	0.14	6.96
32	0.014	0.09	11.4
64	0.009	0.06	17.8

V. RELATED WORK

Previously published X10 codes have been limited to the NPB 3.0 and HPCC benchmarks[28], [29]. Although these benchmarks specify a challenging set of computational and memory-intensive codes, they do not exercise all the language features necessary to develop a typical scientific application. In particular, our codes feature complex data structures using distributed arrays, multi-step algorithms and load-balancing strategies that are not requirements of the HPCC benchmarks. Bernholdt et al.[21] explored the programmability of various HPCS languages towards expressing the Hartree-Fock kernel. However, no concrete implementation of the same was provided due to the immaturity of the language at the time of writing their paper.

VI. Concluding Remarks

Writing non-trivial applications in a language under development is not an easy task. However, we feel that the investment of time in attempting to develop scientific applications in an emerging language is essential to help the language mature. This is an especially important task to undertake in a language that is specifically targeted towards HPC applications. Integrating the efforts of application programmers and language designers in such a codesign process is now recognized as fundamental to achieving performance in the exascale era[30].

We found that expressing solutions to non-trivial problems in X10, one such language considered for this work, was intuitive and more readable than writing it in sequential languages with MPI or OpenMP support for parallelism. Our initial experiences in developing these applications have been fruitful for the language developers and have generated substantial improvements in the language implementation. We have also highlighted some of the scenarios where a straightforward implementation (as in the case of HF code) might not yield optimal performance and therefore requires tweaking on a per-application basis.

The current implementation of some language features such as *distributed arrays*, *atomic* sections and *collecting finish* are non-optimal, complicating and sometimes obscuring application-level performance issues. The feedback emanating from the applications presented here has led to improvements in the X10 implementation including: *complex* arithmetic, fast local *array*s, scaling of *distributed arrays* and *async*. As the X10 implementation matures, we plan to continue to improve these applications and work towards future performance analysis of these codes on much larger systems. It is hoped that the lessons learned here are useful for other application and language developers in similar programming systems.

The application codes discussed in this paper are available at http://cs.anu.edu.au/~Josh.Milthorpe/anuchem.html and are free software under the Eclipse Public License.

Acknowledgements

We would like to thank Igor Peshansky and Vijay Saraswat from IBM T.J. Watson Research Center for help with language design and implementation issues, and Daniel Frampton and Steve Blackburn from the ANU for helpful discussions. We would also like to thank Andrew Haigh, who implemented the rotation-based operators for the FMM code. This work was partially supported by the Australian Research Council and IBM through Linkage grant LP0989872.

References

[1] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. Hollingsworth, and M. Zelkowitz, “HPC programmer productivity - a case study of novice HPC programmers,” in *Proceedings of SC ’05*, Nov 2005.

[2] HPCS home page: http://www.highproductivity.org.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioğlu, C. von Praun, and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing,” in *Proceedings of OOPSLA ’05*. ACM, 2005, pp. 519–538.

[4] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and A. Yelick, “The landscape of parallel computing research: A view from Berkeley,” University of California, Berkeley, Tech. Rep., Dec 2006.

[5] L. Greengard and V. Rokhlin, “A new version of the Fast Multipole Method for the Laplace equation in three dimensions,” *Acta Numerica*, vol. 6, p. 229, 1997.

[6] L. Ying, G. Biros, D. Zorin, and H. Langston, “A new parallel kernel-independent fast multipole method,” in *Proceedings of SC ’03*, 2003, p. 14.

[7] C. White and M. Headgordon, “Rotating around the quartic angular momentum barrier in fast multipole method calculations,” *The Journal of Chemical Physics*, vol. 105, no. 12, Nov 1996, 10.1063/1.472369.

[8] H. Dachsel, “Fast and accurate determination of the Wigner rotation matrices in the fast multipole method,” *The Journal of Chemical Physics*, vol. 124, no. 14, p. 144115, Apr 2006, 10.1063/1.2194548.

[9] C. Lambert, T. Darden, and J. Board, “A multipole-based algorithm for efficient calculation of forces and potentials in macroscopic periodic assemblies of particles,” *J. Comput. Phys.*, vol. 126, no. 2, p. 274, Jul 1996.

[10] K. Kudin and G. Scuseria, “A fast multipole method for periodic systems with arbitrary unit cell geometries,” *Chem. Phys. Lett.*, vol. 283, no. 1-2, p. 61, Jan 1998.

[11] U. Essmann, L. Perera, M. Berkowitz, T. Darden, H. Lee, and L. Pedersen, “A smooth particle mesh Ewald method,” *J. Chem. Phys.*, vol. 103, no. 19, pp. 8577–8593, 1995.

[12] A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry*. McGraw-Hill, New York, 1989.

[13] L. E. McMurchie and E. R. Davidson, “One- and two-electron integrals over Cartesian Gaussian functions,” *J. Comput. Phys.*, vol. 26, p. 218, 1978.

[14] H. P. Lüthi, J. E. Mertz, M. W. Feyereisen, and J. E. Almlöf, “A coarse-grain parallel implementation of the direct SCF method,” *J. Comput. Chem.*, vol. 13, p. 160, 1991.

[15] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi, *GNU Scientific Library Reference Manual (3rd Ed.)*. Network Theory Ltd; ISBN 0954612078, 2009.

[16] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[17] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *Proceedings of SC '09*, 2009.

[18] P. Fortin, "Algorithmique hiérarchique parallèle haute performance pour les problèmes à N-corps," Ph.D. dissertation, L'Université de Bordeaux I, Nov 2006.

[19] "MPI-2: Extensions to the message-passing interface," University of Tennessee, Knoxville, Tech. Rep., Nov 2003. [Online]. Available: http://www.mpi-forum.org/docs/mpi2-report.pdf

[20] M. Eleftheriou, J. Moreira, B. Fitch, and R. Germain, "A volumetric FFT for BlueGene/L," in *Proceedings of HiPC '03*, 2003, pp. 194–203.

[21] A. G. Shet, W. R. Elwasif, R. J. Harrison, and D. E. Bernholdt, "Programmability of the HPCS languages: A case study with a quantum chemistry kernel," in *Proceedings of IPDPS '08*. IEEE, 2008, pp. 1–8.

[22] V. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *Proceedings of PPoPP 11*, Feb 2011.

[23] X10 home page: http://x10.codehaus.org/.

[24] H. Berendsen, J. Grigera, and T. Straatsma, "The missing term in effective pair potentials," *J. Phys. Chem.*, vol. 91, no. 24, pp. 6269–71, Nov 1987.

[25] O. Coulaud, P. Fortin, and J. Roman, "High performance BLAS formulation of the adaptive fast multipole method," *Mathematical and Computer Modelling*, vol. 51, no. 3-4, p. 177, Feb 2010.

[26] P. Gill and J. Pople, "The Prism algorithm for Two-Electron Integrals," *Int. J. Quantum Chem.*, vol. 40, pp. 753–772, 1991.

[27] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," *J. Chem. Theory and Computation*, vol. 4, no. 3, pp. 435–447, Mar 2008.

[28] T. Wen, "Introduction to the X10 implementation of NPB MG," IBM, Tech. Rep., Dec 2006. [Online]. Available: http://x10-lang.org/Benchmarks

[29] G. Almási, D. Cunningham, D. P. Grove, I. Peshansky, O. Tardieu, G. Bikshandi, G. Dózsa, S. B. Kodali, V. Saraswat, E. Tiotto, C. Caşcaval, M. Farreras, N. Nystrom, and S. Sur, "IBM's 2009 submission to the HPC Challenge Class 2 Competition," IBM, Tech. Rep., Oct 2009. [Online]. Available: http://x10.codehaus.org/hpcc09

[30] A. Geist and S. Dosanjh, "IESP exascale challenge: Co-design of architectures and algorithms," *Int. J. High Perf. Comput. App.*, vol. 23, no. 4, pp. 401–402, Nov 2009.