Programmability of the HPCS Languages: A Case Study with a Quantum Chemistry Kernel*

Aniruddha G. Shet, Wael R. Elwasif, Robert J. Harrison, and David E. Bernholdt

Oak Ridge National Laboratory
PO Box 2008, Oak Ridge, TN 37831 USA
{shetag,elwasifwr,harrisonrj,bernholdtde}@ornl.gov

Abstract

As high-end computer systems present users with rapidly increasing numbers of processors, possibly also incorporating attached co-processors, programmers are increasingly challenged to express the necessary levels of concurrency with the dominant parallel programming model, Fortran+MPI+OpenMP (or minor variations). In this paper, we examine the languages developed under the DARPA High-Productivity Computing Systems (HPCS) program (Chapel, Fortress, and X10) as representatives of a different parallel programming model which might be more effective on emerging high-performance systems. The application used in this study is the Hartree-Fock method from quantum chemistry, which combines access to distributed data with a task-parallel algorithm and is characterized by significant irregularity in the computational tasks. We present several different implementation strategies for load balancing of the task-parallel computation, as well as distributed array operations, in each of the three languages. We conclude that the HPCS languages provide a wide variety of mechanisms for expressing parallelism, which can be combined at multiple levels, making them quite expressive for this problem.

1 Introduction

As trends in high-performance computing hardware move rapidly towards very large numbers of processor cores, the developers of software for such systems face increasing challenges in producing applications that can effectively use the highest-end resources available. The dominant parallel programming model in current use involves a sequential language (such as Fortran), combined with a two-sided message passing library (such as MPI), and possibly a threading library (such as OpenMP). The continuing viability of this approach in the face of hardware trends is increasingly the subject of debate within the computational science community, with one recent report concluding that "it is virtually certain that MPI will not be able to provide all of the required concurrency" [21].

Among the many efforts to develop new and improved parallel programming models and languages, the High-Productivity Computing Systems (HPCS) program, sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA), deserves particular note. The three new programming languages developed under the DARPA HPCS program (Chapel [1, 12], Fortress [4, 9], and X10 [6, 14]) represent perhaps the largest concerted investment in the development of new environments for parallel programming in several decades. These "HPCS languages" incorporate the results of past research in parallel programming models with novel ideas and approaches relevant to emerging hardware architectures to produce high-level languages with support for object oriented and generic programming, a broad range of constructs for expressing both task and data parallelism at multiple levels, and a global view of data. We (the authors of this paper) believe that these languages are at least *representative* of a new generation of parallel programming environments which are appropriate for widespread use in high-end scientific and technical computing, and therefore worthy of deeper examination.

Our overall methodology is to distill key aspects of various scientific applications into model computations that can be expressed in the HPCS languages. Our focus is on *programmability*, i.e. the mechanisms these languages provide to express the model computations, and comparisons with the traditional message-passing model and other approaches. At present, the language implementations are not

*This work has been supported by the Advanced Research and Development Agency, the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), and the ORNL Postmasters Research Participation Program which is sponsored by ORNL and administered jointly by ORNL and by the Oak Ridge Institute for Science and Education (ORISE). ORNL is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725. ORISE is managed by Oak Ridge Associated Universities for the U. S. Department of Energy under Contract No. DE-AC05-00OR22750.

sufficiently mature to allow a meaningful examination of performance issues. We expect to place more emphasis on the interactions between programmability and performance in future work as the languages progress. The goal of our work is to better understand the capabilities and features of the HPCS languages from the standpoint of computational scientists who may soon *need* to move towards languages of this type in order to continue using the large-scale computer systems available in an effective manner.

In this paper, we consider a kernel from the Hartree-Fock self-consistent field method [15], widely used in quantum chemical simulations, which combines access to distributed data with a task-parallel algorithm and which exhibits irregularity in both data distribution and parallel tasks. Scalable implementation of this algorithm is, at best, extremely challenging in a traditional message-passing model, and was a significant motivation for the development of the Global Arrays Toolkit (GA) [22], which shares some of the core features of the HPCS languages. Key features of the algorithm which will be examined in this paper include dynamic load balancing, and high-level operations on distributed arrays.

The remainder of this paper is organized as follows. In Section 2 we describe the Hartree-Fock problem and a scalable algorithm for it in more detail. In Section 3 we give a brief overview of the HPCS languages, focusing on the features relevant to the Hartree-Fock problem. In Section 4 we explore the concepts the various languages provide to express the load balancing and array operations required by the algorithm. Finally, in Section 5 we summarize our findings and describe our plans for future work with the HPCS languages.

2 The Problem

We have chosen Hartree-Fock method of quantum chemistry [15] as an exemplar of applications combining distributed data, task parallelism, and significant task irregularity.

The most computationally intensive step in the Hartree-Fock method is construction of the Fock matrix,

$$F_{\mu\nu} \leftarrow D_{\lambda\sigma}\{2(\mu\nu|\lambda\sigma) - (\mu\lambda|\nu\sigma)\} \qquad (1)$$

where the indices μ, ν, λ, σ denote the basis functions, and D is the density matrix. $(\mu\nu|\lambda\sigma)$ is a rank-4 tensor representing the two-electron repulsion integrals. Formation of the Fock matrix is an $O(N^4)$ operation for N basis functions.

The basis functions are grouped into electronic shells, and then into atomic centers, based on characteristics of the molecule and basis. The two-electron repulsion integrals are evaluated in blocks based on the shell structure of the basis. The resulting "shell blocks" of the integral tensor vary in size from 1 to more than 10,000 elements. Separately, the computational costs of the integrals also vary over several orders of magnitude and they are not readily predicted in advance.

A scalable parallel implementation of the algorithm requires that both the data (Fock and density matrices) and the computation (integral block evaluation and their contributions to the Fock matrix) be fully distributed. The first such implementation of the Hartree-Fock method was done by Furlani and King [17] using MPI two-sided messaging, but they concluded that the dynamic load balancing required to achieve scalability was too hard to express in MPI, even for small processor counts (at that time, $O(10)$) [16].

Furlani and King's approach was a major motivation for the development of the Global Arrays Toolkit (GA) [22], a library-based parallel programming environment providing a global view of memory with one-sided access, and a few basic parallel programming constructs, such as locks and atomic read-and-increment counters. Use of the GA library enabled the first scalable fully distributed Hartree-Fock implementation [15, 18, 24]. The essence of the algorithm can be summarized as follows:

1. D and the two constituents of F known as the Coulomb (J) and exchange (K) matrices (corresponding to the two terms in Eq. 1) are created as two-dimensional $N \times N$ distributed arrays.

2. Construction of the J and K matrices, per Eq. 1, takes place in a four-fold loop nest over the basis function indices. Due to permutational symmetries among the indices of the two-electron integrals, restrictions are imposed on loop bounds, yielding a triangular iteration space of roughly $\frac{1}{8}N^4$ elements, one eighth the size of the full space. The four-fold loop is typically stripmined, with a granularity chosen as a compromise between the reuse of D, J, and K and load balance. In this work we assume, without loss of generality, that the loop nest is stripmined at the atomic level. Since the tasks are highly irregular in cost, dynamic load balancing is required. (In the GA implementation, an atomic read-and-increment counter is used to allocate tasks to processes as they become available.)

3. In each task, an atomic quartet of integrals is evaluated on the fly. Once computed, an integral is contracted with six different D values and contributes to six different J and K values. The appropriate D, J, and K blocks are cached and reused wherever possible to reduce network traffic. All tasks are independent, except for the updates to the J and K matrices.

4. Finally, the J and K matrices must be symmetrized and combined to form F, which can be done in a data-parallel fashion. (The GA library provides basic linear algebra operations on the distributed arrays, including transposition.)

3 Overview of Languages

At a high level, the three languages have many similarities, though in detail there are significant differences. All three languages emphasize the expression of parallelism at a high level by the programmer and relying on the compiler/runtime/library infrastructure to produce an optimized implementation for the underlying parallel architecture.

In all three, program execution starts with a single conceptual thread of control, which then generates more parallelism through the use of language constructs (i.e. *not* strictly SPMD). Parallelism is mapped onto a multi-level conceptual model that is *roughly* approximated by the "processes" and "threads" of the traditional MPI-based programming model, for which each language has different terminology. Memory in all three is globally addressable, and data is global and can be distributed. Locality control permits computation and data to be assigned to specific system resources for performance reasons. The base languages are object-oriented and provide generic programming capabilities.

3.1 Chapel

Chapel is being designed by Cray Inc. to support general parallel programming while narrowing the gap between mainstream and HPC languages. Chapel's design builds on concepts from ZPL [8, 13], High-Performance Fortran [19], and Cray's multithreaded extensions to C and Fortran, while adopting a variety of other useful features from mainstream and academic languages.

A *locale* in Chapel symbolizes a unit of architectural locality on the target machine, containing processing and storage capabilities. A locale's memory is uniformly accessible to computations running on it. Each locale supports a dynamic set of tasks that are created using *begin*, *cobegin*, and *coforall* statements. Tasks are synchronized using synchronization (*sync*) variables that have full/empty semantics, and *atomic* sections that provide transactional memory capabilities. Data and tasks can be mapped to machine resources (locales) using *on* clauses. The mapping may be explicitly specified, or data driven. Chapel supports data parallelism via *domains*, a first-class language concept representing an index set. Domains can be iterated over in parallel using *forall* and *coforall* loops, and are used to declare, resize, and slice arrays. Domains and their arrays may be partitioned across a set of locales using *distributions*, which map from the global view of an aggregate to its implementation on distinct locales.

3.2 Fortress

Fortress, being developed by Sun Microsystems, Inc., is designed to be an open, growable language. Consequently, it is designed with a small set of core language features, and the majority of concepts are coded in libraries.

Fortress programs are multithreaded; a user may explicitly spawn *threads*, or call *implicitly parallel* constructs that create threads managed by the Fortress language implementation. *Atomic* sections enable synchronization of threads. Fortress *regions* abstractly describe the underlying machine structure and can have an arbitrary hierarchical structure. Thread affinity to particular regions may be specified with *at* expressions, and *distributions* allow management of data locality. Parallelism can be programmed inside libraries as *distributions* and *generators*.

Fortress also provides a variety of novel features targeting the HPCS program's productivity goals, including built-in constructs for managing components and interfaces, expressing tests and contracts, and methods for rendering the code that look like typeset mathematics.

3.3 X10

X10, which is being developed by IBM Corp., is designed to leverage the extensive software ecosystem around the Java language. X10 is defined as a serial subset of Java, extended with additional concurrency, distribution, and locality features.

In X10, a *place* corresponds to a data-coherent processing element, with each *place* supporting a dynamic set of lightweight *activities*. Activities specify logical parallelism and may be composed in arbitrarily nested ways using *async*, *future*, *foreach*, and *ateach* constructs, and are translated by the X10 compiler/runtime into running threads. An activity executes to completion on the place where it is created, but can launch activities on other places, and detect termination of all such activities via the *finish* statement. *Clocks* enable synchronization of dynamically created activities across places. Activities within a place uniformly and coherently access its memory using *atomic* statements; weaker ordering semantics exist for inter-place data accesses. Similar to Chapel, X10 provides a ZPL-like "array language" to express high-level operations on distributed arrays.

3.4 Language Versions and Limitations

Versions of the languages used in this paper are shown in Table 1. The Fortress and Chapel implementations have thus far focused on the multi-threading capabilities of the languages, and neither supports explicitly multi-processor code at this time. Where these issues prevent us from actually implementing certain approaches, we may, for the sake of completeness, discuss *proposed* implementations based on the language specifications.

4 Programming Examples and Discussion

With their rich parallel semantics, the HPCS languages offer a variety of ways to implement algorithms like Fock matrix construction. In this section, we present and discuss examples from various strategies we have developed in the three languages.

Table 1. Language Versions

Language	Specification	Implementation
Chapel	v0.750 [2]	v0.5.375 compiler
Fortress	v1.0alpha [9]	v0.1 alpha interpreter
X10	v1.1 [7]	v1.5 compiler

Code 1 Static, Program Managed Load Balancing - X10

```
1 place placeNo = place.FIRST_PLACE;
2 finish for(point [iat] : [1:natom])
3   for(point [jat, kat] : [1:iat, 1:iat])
4     for(point [lat] :
5                [1:(kat==iat?jat:kat)]) {
6       async (placeNo)
7         buildjk_atom4(new blockIndices(...));
8       placeNo = placeNo.next();
9     }
```

Sections 4.1– 4.4 present different load-balancing strategies for the four-fold loop in the Fock matrix construction (step 2 in the description in Section 2). In Section 4.5 we examine how the languages support various kinds of operations on distributed global-view arrays required in steps 1, 3, and 4 of the Fock matrix construction.

Except where limited by the current implementations of the languages (see Section 3.4), we have implementations of the strategies presented here for each of the three languages, however due to space limitations, we show code fragments for only one of the implementations of each strategy. A more complete presentation is available in a technical report [23], and the complete set of codes can be obtained from the authors.

4.1 Static, Program Managed Load Balancing

We begin with a statically distributed non-scalable implementation to illustrate how the HPCS languages differ from more familiar SPMD environments in *creating* parallelism. By "program managed" we mean the programmer controls the allocation of work to processors.

4.1.1 X10

Code 1 shows the X10 implementation of a simple round-robin workload distribution to processors. An X10 program starts as a single *root activity*, on the *first place*, and iterates through the four-fold loop (lines 2–5). Note that the loop indices are of type `point`, which is associated with the specified index space, rather than simple integers, as they would be in a traditional programming environment, thus providing a higher degree of type-safety.

In each iteration of the four-fold loop, the *root activity* launches an activity to asynchronously evaluate the task (in all presented code segments, `blockIndices` is a *class* whose member arguments specify the work to be performed in one task) on the remote place specified by `placeNo` (lines 6–7). Then `placeNo` is updated to the next value in the cyclically ordered set of places (line 8), and the *root activity* continues with the next iteration.

Code 2 Dynamic, Language Managed Load Balancing - Fortress

```
1 for iat<-1#natom, jat<-1#iat, kat<-1#iat,
2     lat<-1#(if (kat=iat) then jat
3                          else kat end) do
4   buildjk_atom4 blockIndices(...)
5 end
```

The `finish` construct placed at the outermost level of the loop nest (line 2) forces the *root activity* to await the termination of `async` activities launched within its scope (in this case the four-fold loop). This ensures that all parallel tasks are completed before proceeding.

4.1.2 Chapel

Chapel allows users to specify *iterators* that produce a set of points in an index space with a specified distribution across *locales*. In our proposed multi-*locale* code, the iterator would involve the four-fold loop with an "`on` *locale* `yield` *block*" statement to return the iterator's results, where *block* represents the quartet of indices. The `on` construct would result in the iterator yielding different blocks for different *locales*. A `forall` loop driven by this iterator would process each point on the *locale* where they were yielded by the iterator.

4.1.3 Fortress

Our proposed multi-*region* Fortress implementation would be very similar to the Chapel approach, using the *generator* concept. The generator would feed a parallel *for* construct performing its iterations according to the placement of indices from the generator.

4.2 Dynamic, Language Managed Load Balancing

The simplest possible scalable implementation would be if the language runtime could be relied upon to take care of the load balancing without the programmer even needing to express it in code. It is important to note that such capabilities are current research topics for the languages. Therefore, we present this approach to illustrate the potential for extreme simplicity, but with the caveat that it is still quite speculative.

4.2.1 Fortress

The Fortress `for` construct (Code 2, line 1) is parallel by default and, driven by the loop's *generator*, would (conceptually) spawn a new *thread* for each point in the iteration space (line 4). The Fortress specification anticipates that the runtime will be able to load balance computations that expose substantially more parallelism than the available processors. Fortress has a fairly powerful *generator* concept that allows the entire four-fold loop to be expressed in a single statement. (Loops and generators with explicit sequential semantics are possible too.)

4.2.2 Chapel

Chapel provides *distributions* as a mechanism for distributing an index space (*domain*) across *locales*. *Distributions* may be written to dynamically divide indices among *locales*. A *forall* looping on such a distributed domain would be a way of achieving dynamic load balancing. The feasibility of building this feature into Chapel and its application to the Fock algorithm is an open research issue at present.

4.2.3 X10

The X10 specification requires that data and *activities* remain in the *place* they were created or spawned for their lifetime. However, X10 *places* are virtual, so that many *places* might be mapped to each physical processor, and conceivably migrated among them by the runtime for load balancing and other resource management purposes, similar to Cilk's work stealing [3, 11] within an SMP node or CHARM++ [5, 20] in the distributed context. Given a runtime with such a capability, the simplest X10 implementation would be nearly identical to Code 1, but with many more *places* than processors, so that one or a few atom blocks were allocated to each *place*.

4.3 Dynamic, Program Managed Load Balancing Using a Shared Counter

A dynamically load balanced computation involves all participating processors (conceptually) sharing a single list of tasks, and whenever a processor is free, it takes another task from the list. One common approach, and the one we use here, to implementing the shared task list is to have all processors locally generate tasks in the same sequence, and use a globally shared counter (typically implemented with an atomic read-and-increment operation) to track how many tasks have been taken by processors.

4.3.1 X10

In Code 3, the *root activity* creates the globally shared counter `G` on the *first place* (line 1). Then it uses the `ateach` construct (line 2) to launch a copy of the Fock-build algorithm (lines 4–20) on each *place*. The `finish` (line 2) causes the *root activity* to block until the rest of the algorithm completes on every *place*.

Each *place* iterates over the same sequence of tasks (the four-fold loop, lines 8–11), using `L` to count the tasks. When `L` matches the next task assigned to the *place* (`myG`), it evaluates that integral block. Assignments (`myG`) are obtained from a remote atomic read-and-increment operation on the globally shared counter `G` on the *first place* (lines 5–7, 13–14 and 16). When every *place* has completed the four-fold loop, all tasks will be evaluated.

X10 requires that remote references to mutable data (in this case the shared counter `G`) be done asynchronously, hence the use of the `future` construct at lines 5 and 13.

Code 3 Shared counter for dynamic load balancing - X10

```
1  int G = 0;
2  finish ateach(point [p] :
3         dist.factory.unique(place.places)) {
4    int myG, L = 0;
5    future<int> F = future (place.FIRST_PLACE)
6                    {read_and_increment_G()};
7    myG = F.force();
8    for(point [iat] : [1:natom])
9      for(point [jat, kat] : [1:iat, 1:iat])
10       for(point [lat] :
11                 [1:(kat==iat?jat:kat)]) {
12         if (L == myG) {
13           F = future (place.FIRST_PLACE)
14               {read_and_increment_G()};
15           buildjk_atom4(new blockIndices(...));
16           myG = F.force();
17         }
18         ++L;
19       }
20 }
```

Code 4 Atomic read-and-increment - X10

```
1 private int read_and_increment_G() {
2   int myG;
3   atomic myG = G++;
4   return myG;
5 }
```

Separation between spawning the future and forcing it (as in lines 13 and 16) allows computation and communication to be overlapped.

Code 4 shows how the atomic read-and-increment operation is straightforwardly implemented with an `atomic` section.

4.3.2 Chapel

Our Chapel implementation employs the `coforall` statement to create distinct concurrent computations for all the *locales*. (The Chapel `forall` construct only specifies that the iterations *may* run concurrently, while `coforall` requires a separate computation for each iteration.)

The shared counter `G` is defined as a *synchronization variable* using the `sync` type, which provides "full/empty" semantics. Once written, such a variable cannot be rewritten until it is emptied. Likewise, an empty variable cannot be re-read until it is written. Computations attempting to write to a full `sync` variable or read from an empty one will block until another computation changes the variable's state. Taking advantage of these semantics to atomically update the counter, every computation first does a read followed immediately by a write of `G` to fetch the next task. The processing of a newly assigned task is overlapped with the fetch of the next task using a `cobegin` statement.

4.3.3 Fortress

In the Fortress version, the *for* expression spawns a thread for each *region*. An implicitly parallel *tuple* expres-

Code 5 Task pool of integral blocks - Chapel

```
1  class taskpool {
2    const poolSize;
3    var taskarr : [0..poolSize-1]
4                  sync blockIndices;
5    var head, tail : sync int = 0;
6    def add(blk : blockIndices) {
7      const pos = tail;
8      tail = (pos+1)%poolSize;
9      taskarr(pos) = blk;
10   }
11   def remove() {
12     const pos = head;
13     head = (pos+1)%poolSize;
14     return taskarr(pos);
15   }
16 }
```

Code 6 Top-level driver for task pool - Chapel

```
1 config const numConsumers = 10,
2               poolSize = numConsumers;
3 const t = taskpool(poolSize);
4 cobegin {
5   coforall cons in 1..numConsumers do
6     consumer();
7   producer();
8 }
```

sion runs a new task concurrently with updates to the shared counter `G`. The `read_and_increment_G` function is implemented as an `atomic` method in Fortress.

4.4 Dynamic, Program Managed Load Balancing Using a Task Pool

The task pool model of dynamic load balancing uses common work area, or "pool" into which producers submit tasks, and consumers remove and execute them. This is a general pattern of synchronization applicable to a wide variety of problems.

4.4.1 Chapel

In Chapel, the pool of integral block tasks is built around an array of `sync` variables `taskarr` (Code 5, lines 3–4). Methods are defined for producers to add tasks to the pool (lines 6–10) and for consumers to remove them (lines 11–15).

The main application (Code 6) begins as a single computation and sets up a task pool on the first *locale*. For simplicity of presentation, we have arbitrarily fixed both the number of consumers (`numConsumers`) and the size of the task pool (`poolSize`), but `config const` allows these to be

Code 7 Producer of integral blocks - Chapel

```
1 def producer() {
2   forall blk in genBlocks() do
3     t.add(blk);
4 }
```

Code 8 Fock index space iterator - Chapel

```
1  def genBlocks() {
2    forall iat in 1..natom do
3      forall (jat, kat) in [1..iat, 1..iat] {
4        const lattop = if (kat==iat) then jat
5                                    else kat;
6        forall lat in 1..lattop do
7          yield blockIndices(...);
8      }
9    forall cons in 1..numConsumers do
10     yield nil;
11 }
```

Code 9 Consumer of integral blocks - Chapel

```
1  def consumer() {
2    var blk = t.remove();
3    while (blk != nil) {
4      const copyofblk = blk;
5      cobegin {
6        buildjk_atom4(copyofblk);
7        blk = t.remove();
8      }
9    }
10 }
```

specified at the time of running the application. In practice these would be derived from other variables like the count of *locales* and cores, and the granularity of tasks (assumed here to be 1). The `cobegin` (line 4) runs the producer (line 7) and consumer (lines 5–6) computations in parallel. The `coforall` construct (line 5) guarantees that the loop iterations run concurrently.

The producer (Code 7) simply adds atomic quartets to the task pool. The atomic quartets come from the `genBlocks` iterator (Code 8), which steps through the fourfold loop (lines 2–6) and then generates sentinel values (lines 9–10) to signal the consumers that there are no more tasks.

Consumers (Code 9) take tasks from the pool and evaluate them until the sentinel value described above is encountered. The `cobegin` construct (line 5) allows the integral evaluation task to be overlapped with obtaining the next atomic quartet.

4.4.2 X10

The X10 implementation is very similar to the Chapel version. The *root activity* instantiates the task pool on the *first place*, spawns consumer activities on all *places*, and then runs the producer activity. The X10 task pool implementation uses *conditional atomic* sections to coordinate the interacting activities.

4.4.3 Fortress

Our proposed implementation in Fortress would use features like *for* and *also do* to enable producer and consumer *threads* to run together. The producer would be driven

Figure 1. Array Functionality

Operations		Language constructs used		
		Chapel	Fortress	X10
Mixed data and task parallelism		cobegin (task) + forall loop (data)	tuple (task) + for loop (data)	finish async (task) + ateach (data)
Global-view array operations	initialization	array initialization expressions	function expressions	array initialization functions
	arithmetic	array promotions of scalar operators (+,*)	fortress library operators (+,juxtaposition)	array class methods (add,scale)
	sub-array	slicing	array factory function (subarray)	restriction

Code 10 Symmetrization of J and K - Chapel

```
1 cobegin {
2   [(i,j) in D] jmat2T(i,j) = jmat2(j,i);
3   [(i,j) in D] kmat2T(i,j) = kmat2(j,i);
4 }
5 jmat2 = 2*(jmat2+jmat2T);
6 kmat2 += kmat2T;
```

by a *generator*. The task pool implementation would use *abortable atomic* expressions, which allow atomic sections to validate conditions and rollback on violations.

4.5 Multi-Dimensional Array Functionality

All three languages provide a rich set of global array functionality including physical distribution, initialization, one-sided put and get accesses, and data parallel algebraic operations. The array functionality used in our Fock build codes is captured in figure 1. We delve into details of the formation of the final Fock matrix from the computed Coulomb (J) and exchange (K) matrices to illustrate how each language expresses some of these operations.

4.5.1 Chapel

In Chapel, *forall* expressions are used to transpose the two matrices (Code 10, lines 2–3). Both the arrays and their transposes are defined over the *domain* (index space) `D` (definitions not shown). The loop indices `i` and `j` are drawn from the index space `D`, and as with a normal `forall` they may be done in parallel. The `cobegin` (line 1) allows the two transpositions to be carried out in parallel as well. Lines 5–6 illustrate how Chapel promotes scalar operators to apply to arrays.

4.5.2 Fortress

The *tuple* expression in Code 11 line 1 spawns separate *threads* to evaluate its elements. `t()` is an array factory method that computes an array transpose by iterating over the array indices in an implicitly parallel *for* loop. In lines 2 and 3, the Fortress library operators + and juxtaposition (multiplication) are applied to arrays.

Code 11 Symmetrization of J and K - Fortress

```
1 (jmat2T,kmat2T) = (jmat2.t(),kmat2.t())
2 jmat2 := 2(jmat2+jmat2T)
3 kmat2 := kmat2+kmat2T
```

Code 12 Symmetrization of J and K - X10

```
1  finish {
2    async ateach(point[i,j] : D)
3      jmat2T[i,j] = future (D[j,i])
4                    {jmat2[j,i]}.force();
5    async ateach(point[i,j] : D)
6      kmat2T[i,j] = future (D[j,i])
7                    {kmat2[j,i]}.force();
8  }
9  jmat2 = jmat2.add(jmat2T).scale(2);
10 kmat2 = kmat2.add(kmat2T);
```

4.5.3 X10

Code 12 shows a naïve transposition in X10. This implementation launches a separate asynchronous *activity* for each element of the matrix (points in the *distribution* `D`) (lines 2 and 5). *Futures* are launched on the *place* holding the `[j,i]` element of the index space to retrieve the remote value. The surrounding `finish` (line 1) ensures completion of the two transpositions before continuing. `add` and `scale` (lines 9–10) are array class methods.

Note that the transposition can be expressed much more efficiently in X10 (fewer *activities*, better locality, aggregated data movement) [10], though not as succinctly.

5 Conclusions and Future Work

We have presented several possible implementations for load balancing of a computation involving tasks of widely varying cost in each of the HPCS languages, as well as operations on distributed arrays. Though the languages differ in their detailed syntax and semantics, at a higher level, they provide similar capabilities, which generally go well beyond those of the traditional message-passing model, and even the Global Arrays programming model that was used for the first truly scalable, fully-distributed implementation of the Hartree-Fock method. The examples illustrate the wide range of constructs provided for the expression of parallelism, and how they can be combined to expose the maximum possible parallelism to the language. This will be an important aspect of future programming environments as the number of CPUs and processor cores continues to grow rapidly.

Future work includes examination of the performance considerations associated with different implementations, which we plan to undertake once the language implementations have reached an appropriate level of maturity. We

also plan to extend these studies to other scientific applications, in order to illustrate other aspects of the languages.

Acknowledgements

We wish to thank all three of the language development teams for their support and discussions through this work. We would also like to thank Brad Chamberlain for assistance in improving the display of the listed codes.

References

[1] Chapel home page. http://chapel.cs.washington.edu/.

[2] Chapel language specification. http://chapel.cs.washington.edu/spec-0.750.pdf.

[3] Cilk home page. http://supertech.csail.mit.edu/cilk/.

[4] Fortress home page. http://fortress.sunsource.net/.

[5] Parallel objects: CHARM++. http://charm.cs.uiuc.edu/research/charm/.

[6] X10 home page. http://x10.sf.net/.

[7] X10 language specification. http://x10.cvs.sourceforge.net/x10/x10.man/v1.0/x10.pdf?view=log.

[8] ZPL home page. http://www.cs.washington.edu/research/zpl/.

[9] E. Allan, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt. The fortress language specification version 1.0alpha. September 2006. http://research.sun.com/projects/plrg/Publications/fortress1.0alpha.pdf.

[10] G. Bikshandi, J. Castanos, S. Kodali, S. Krishnamoorthy, V. K. Nandivada, I. Peshanshy, V. Sachdeva, V. Saraswat, M. Stephenson, S. Sur, P. Varma, and T. Wen. HPC challenge submission for X10 (sc 2007). http://x10.sourceforge.net/applications/benchmark/HPCC/x10-hpcc07-v3.pdf, 2007.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[12] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.

[13] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and L. Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing (PPHEC)*, 2004.

[14] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA)*, October 2005.

[15] I. T. Foster, J. L. Tilson, A. F. Wagner, R. L. Shepard, R. J. Harrison, R. A. Kendall, and R. J. Littlefield. Toward High-Performance Computational Chemistry: I. Scalable Fock Matrix Construction Algorithms. *Journal of Computational Chemistry*, 17(1):109–123, 1996.

[16] T. R. Furlani. private communication, 1995.

[17] T. R. Furlani and H. F. King. Implementation of a parallel direct SCF algorithm on distributed memory computers. *J. Computat. Chem.*, 16(1):91–104, 1995.

[18] R. J. Harrison, M. F. Guest, R. A. Kendall, D. E. Bernholdt, A. T. Wong, M. Stave, J. L. Anchell, A. C. Hess, R. J. Littlefield, G. L. Fann, J. Nieplocha, G. S. Thomas, D. Elwood, J. L. Tilson, R. L. Shepard, A. F. Wagner, I. T. Foster, E. Lusk, and R. Stevens. Toward High-Performance Computational Chemistry: II. A Scalable Self-Consistent Field Program. *Journal of Computational Chemistry*, 17(1):124–132, 1996.

[19] High Performance Fortran Forum. High performance fortran language specification, version 2.0. Technical report, Rice University, 1997.

[20] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng. Programming petascale applications with Charm++ and AMPI. In D. A. Bader, editor, *Petascale Computing: Algorithms and Applications*, volume 1 of *Chapman & Hall/CRC Computational Science*. Chapman & Hall/CRC, 2007.

[21] R. C. Murphy. Workshop on programming languages for high performance computing (HPCWPL) final report. Technical Report SAND2007-2047, Sandia National Laboratories, 2007. http://www.cs.sandia.gov/CSRI/Workshops/2006/HPC_WPL_workshop/HPCWPL_FinalReport.pdf.

[22] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[23] 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 (extended version). Technical report ORNL/TM-2008/011, Oak Ridge National Laboratory, 2008. See http://www.ornl.gov/info/reports/.

[24] J. L. Tilson, M. Minkoff, A. F. Wagner, R. Shepard, P. Sutton, R. J. Harrison, R. A. Kendall, and A. T. Wong. High-Performance Computational Chemistry: Hartree-Fock Electronic Structure Calculations on Massively Parallel Processors. *International Journal of High Performance Computing Applications*, 13(4):291–302, 1999.