Survey on High Productivity Computing Systems (HPCS) Languages

[Internal Report]

Saliya Ekanayake
School of Informatics and Computing,
Indiana University
sekanaya@cs.indiana.edu

Abstract

Parallel languages have been focused towards performance, but it alone is not be sufficient to overcome the barrier of developing software that exploits the power of evolving architectures. DARPA initiated high productivity computing systems (HPCS) languages project as a solution which addresses software productivity goals through language design. The resultant three languages are Chapel from Cray, X10 from IBM and Fortress from Sun. We recognize memory model (perhaps namespace model is a better term) as a classifier for parallel languages and present details on shared, distributed, and partitioned global address space (PGAS) models. Next we compare HPCS languages in detail through idioms they support for five common tasks in parallel programming, i.e. data parallelism, data distribution, asynchronous remote task creation, nested parallelism, and remote transactions. We conclude presenting complete working code for k-means clustering in each language.

Keywords

High productivity computing systems, Chapel, X10, Fortress

1. Introduction

Parallel programs have been common in scientific analyses as a mean to improve performance or to handle large scales of data. Currently, it is being adopted into other domains as well with the rising data sizes and the availability of computing power in bulk. The steps of forming a parallel program [5] are given in Figure 1.

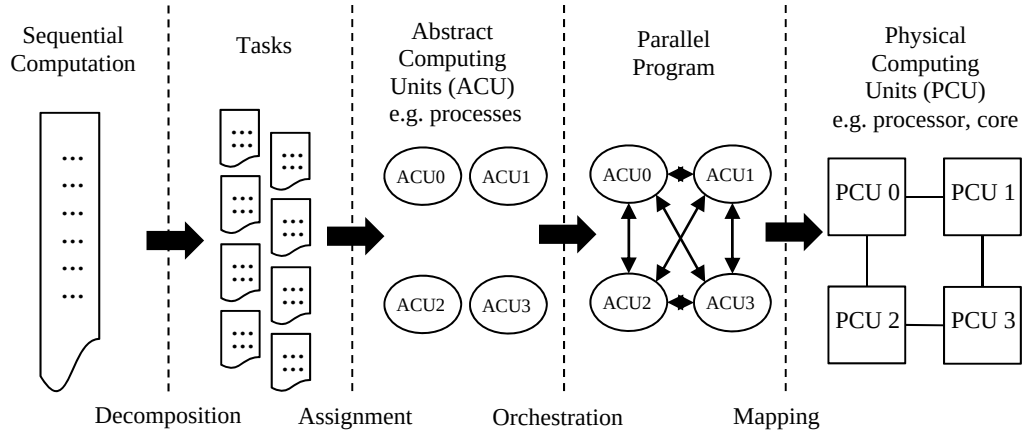

Figure 1. Steps of creating a parallel program

Decomposition generally means to decompose the domain where computations for sub domains may be done in parallel with data exchanged at boundaries. This technique is known as *data parallelism* in the literature [10] and is not be confused with the idea of *pleasingly parallel* applications. In data parallelism the computations on sub domains will be similar to each other. Another form of parallelism that one could exploit in decomposition stage is *task parallelism* [10] where non interdependent different computations are run in parallel.

Once the decomposed tasks are identified, they need to be assigned to abstract computing units (ACU) in the particular parallel programming language or library (hence-forth distinguished from the term "language" only if necessary). Parallel languages provide constructs to create ACUs in one or more of the following forms.

- Explicit Parallel Constructs
  - In this form the programmer explicitly instructs the runtime to execute a designated segment of code in parallel. Spawning a new process using the classical fork system call, `start()` method in `java.lang.Thread` class, `Parallel.Foreach` in Microsoft Task Parallel Library (TPL), and `forall` in Chapel are example constructs of this form.
- Implicit Parallelization
  - This is compiler introduced parallelization for known patterns of code without affecting the semantics of the program. For example, Fortress evaluates segments like tuples, `for` loops, `also do` blocks, and function arguments in parallel.

- Compiler Directives
  - This may be considered as both explicit and implicit forms of parallelization since the programmer explicitly inform the segment to be parallelized while compiler performs the breakdown of it in to parallel tasks. For example, the `#pragma omp parallel for` directive in a C or C++ program based on OpenMP informs the compiler to generate library calls to parallelize the for loop following the directive.

To avoid any confusion, we would like to emphasize here that ACUs do not need to be fixed at start time or execute the same code in a single program multiple data (SPMD) fashion.

Decomposition may result in dependencies, which needs to be handled through some form of communication between ACUs. This may not necessarily be through message passing between ACUs. Globally shared data structures could be used in orchestration as well. In fact, the model of orchestration is specific to the programming model of the language where some may provide fully-fledged communication between ACUs while some may restrict to specific patterns only. The original MapReduce model [6] for example, allows communication between Map and Reduce tasks only. However, it is also worth noting that complex parallel applications may even be developed over less sophisticated programming models. Parallel implementations of latent Dirichlet allocation (LDA) [15] and deterministic annealing multi-dimensional reduction [2] for example are some non-trivial applications developed on the restricted MapReduce model.

The parallel program development is complete after orchestration. The remaining step is to map ACUs to physical computing units (PCU), which is done by the language's runtime. Note the user may have some control over the mapping such as specifying the maximum number of cores to use, etc. but most of the control is up to the runtime.

2. Parallel Programming Memory Models

One classifier of parallel languages is the *memory mode*, which we intended to identify the abstraction of the address space of a program rather the physical composition of memory hence-forth. In fact, the models discussed below may be implemented on either physically shared or distributed memory systems [10]. Three major memory models are being used at present as described in the following sections.

2.1 Shared Memory Model

This model presents the programmer with a shared memory layer to his or her programs. The programmer's view of the program is then a collection of tasks acting on top of a single address space as shown in Figure 2.

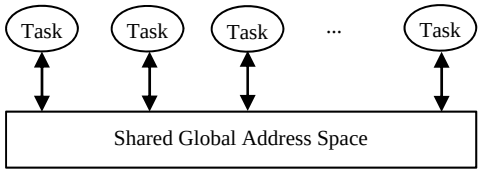

Figure 2. Programmer's view of a shared memory model

The language supporting a shared memory model is responsible for mapping it on to the physical computing system. Usually a task will get mapped to an executable entity in the operating system like a Light Weight Process (LWP). Note. The mapping may not necessarily be one to one and may even take the forms of one to many and many to many. Implementing the shared view of global address space with physically distributed memory is not straightforward and requires communication. Overall the architecture of a shared memory model implementation would be similar to Figure 3. Note. The leftmost processor is enlarged to show the mapping of tasks into central processing units (CPU), and the network layer is depicted between processor and memory of each machine for the sake of clarity and does not imply uniform memory access. Also, though not shown in the figure, tasks can have unshared local address spaces in physical memory local to the processor running the particular task.

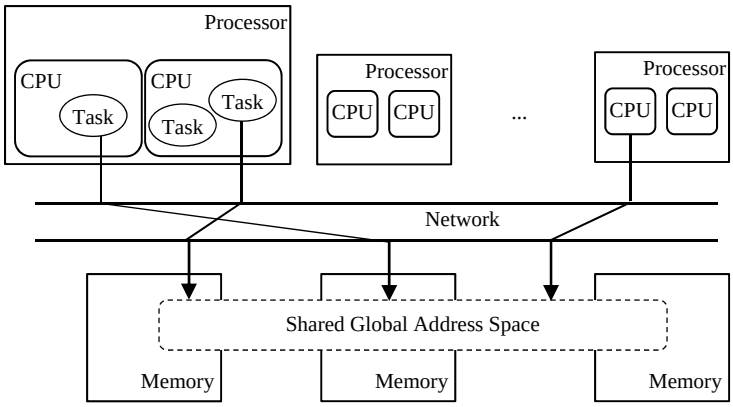

Figure 3. Architecture of shared memory model implementation

Access of a variable that is global to tasks requires transfer of data via network from the residing machine's memory to the requested task. This transfer is made transparent to the programmer by the programming language. Implementations may take the advantage of physically shared memory if exist and may even refrain from supporting distributed memory systems. It is natural in such cases to combine this model with another model supporting distributed memory systems, which coincidently fits well with present day multi-core nodes.

The highlight of this model in a programmer's perspective is the ability to share data among tasks without being constrained by the notion of ownership, which would otherwise result in explicit communication to share data. This property is identified as global view of computing, which is presented in section 3.2. The caveat, however, is the need to synchronize data accesses to preserve the expected behavior of the program.

2.1.1 Open Multi-Processing (OpenMP)

OpenMP is one of most popular implementations of shared memory models. It provides an Application Programming Interface (API) based on compiler directives to express parallelism explicitly and a set of library routines and environment variables. Realizations of this standard supports C/C++ or Fortran languages and portable enough to run on both Unix/Linux and Windows based platforms. OpenMP may not be used directly on distributed memory architectures as it supports only shared memory systems.

An OpenMP program by default follows a fork-join execution model in which a master thread starts at the entry point and threads are forked to execute the denoted parallel regions followed by a join of the threads as depicted in Figure 4.

The number of threads spawned in the parallel region may be decided by the programmer by setting OpenMP execution to static mode and specifying it manually in which case the program is either guaranteed to get that many threads for each parallel region or fail otherwise. The decision on the number of threads may either be left to the operating system by turning on the dynamic execution mode and specifying the maximum number of threads to use if possible, though it may use less than that. In either case, all the threads are numbered from zero to one less than the total number of threads. Thus, it is also possible for different threads to take different paths of executions depending on their thread number inside a parallel region. Finally, OpenMP gives several synchronization constructs, i.e. critical, atomic, barrier, master, ordered, and flush, to the programmer to avoid race conditions when multiple threads work on shared data.

2.1.2 POSIX Threads (Pthreads)

Pthreads is a different yet equally popular implementation of this model. It is a user level library supporting the notion of threads for C programs. Unlike OpenMP, the programmer needs to explicitly encapsulate the code to be run in parallel by a thread. A Pthread program is a single process with threads being spawned along its execution time as intended by the programmer. Threads are separate sequences of execution with no notion of a hierarchy, i.e. parent-child relationship, and dependency between them. Typical execution model of a Pthread program is show in Figure 5.

The program starts with a single thread, *T1*. It continues execution for some time and spawns *T2* and *T3*. All *T1*, *T2*, and *T3* continue to execute until *T1* terminates. *T2* and *T3* continue and spawn more threads down the line. Each horizontal line near the arrow heads indicates the termination of the particular thread, hence the entire program terminates at the end of *T4* according to the diagram.

Joining threads similar to the case with OpenMP is also possible with Pthreads. The *pthread_join* construct, which does this, blocks the calling thread until the other, i.e. joining, thread specified by a *thread-id* terminates. However, it is an error to

attempt multiple joins on the same joining thread. Pthreads also supports two other thread synchronization mechanisms called mutexes and conditional variables. In depth details on these, however, are not in the scope of this paper.

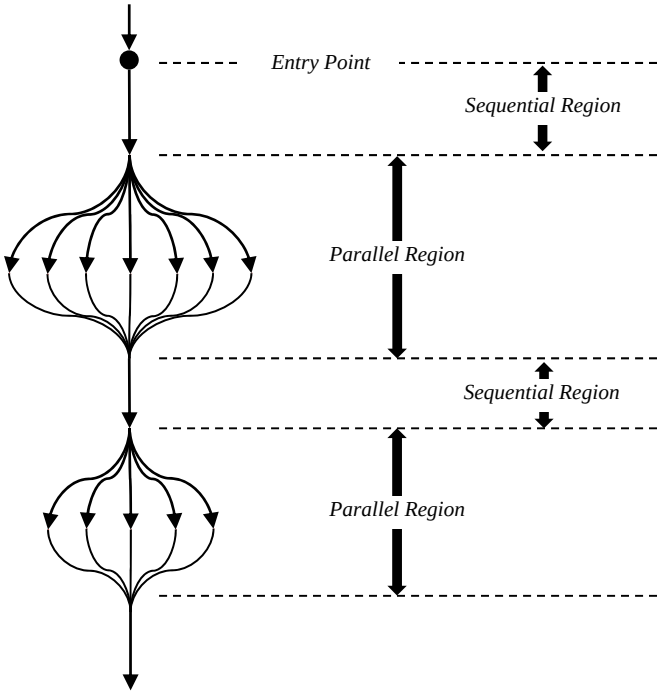

Figure 4. Fork-join execution model of OpenMP

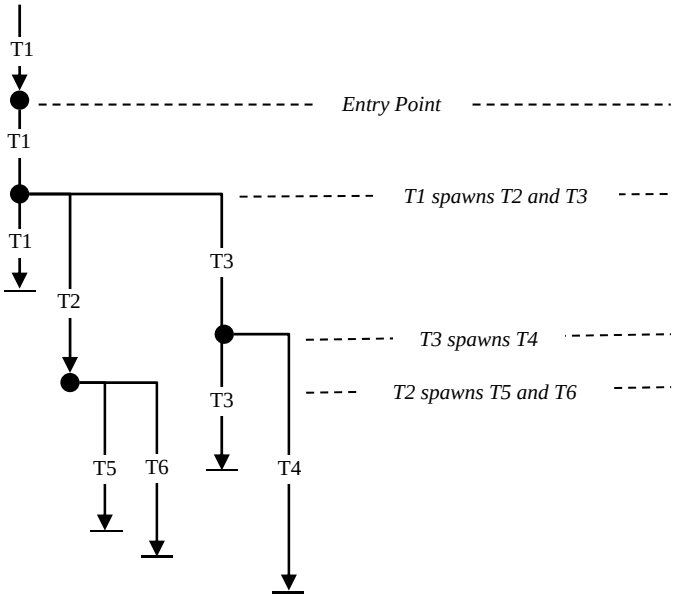

Figure 5. Execution model of a Pthread program

2.2 Distributed Memory Model

Distributed memory model, as it name suggests presents a segmented address space where each portion is local only to a single task. The programmer's view of the program is a collection of individual tasks acting on their own local address spaces as shown in Figure 6. Programmer's view of distributed memory model

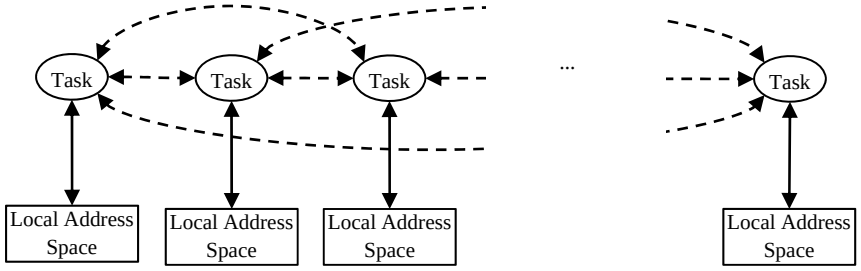

Figure 6. Programmer's view of distributed memory model

In contrast to the view of a program based on shared memory model as shown in Figure 2, this presents a *share nothing* view in terms of address space of the program. This obligates the programmer to partition data structures that need to be distributed among tasks manually. In essence, this model does not support the notion of distributed data structures. Another implication is that tasks need a mechanism to communicate between each other to cooperatively work in parallel as shown in dashed arrows in Figure 6.

The *share nothing* view also encourages a SPMD programming model. Unlike in shared memory model implementations where a single thread starts the execution, multiple tasks start executing the same program independently with SPMD. A task id assigned by the underlying implementation is used to differentiate the flow of execution inside the program.

Mapping of this model on to the physical hardware is fairly straightforward where a task would get mapped to a unit of execution in the operating system as before and the address space local to the particular task would get allocated in the physical memory local to the processor running the task. Figure 7 depicts an overview of the mapping of this model onto physical hardware. The leftmost processor and memory are enlarged to show the mapping of tasks and address spaces.

Accessing data remote to a particular task requires explicit communication over the network as indicated in Figure 7. The programmer is responsible for using the constructs provided by the underlying abstraction to specify such communication. Variables local to a task reside in physical memory of the same machine and require no network communication to access them.

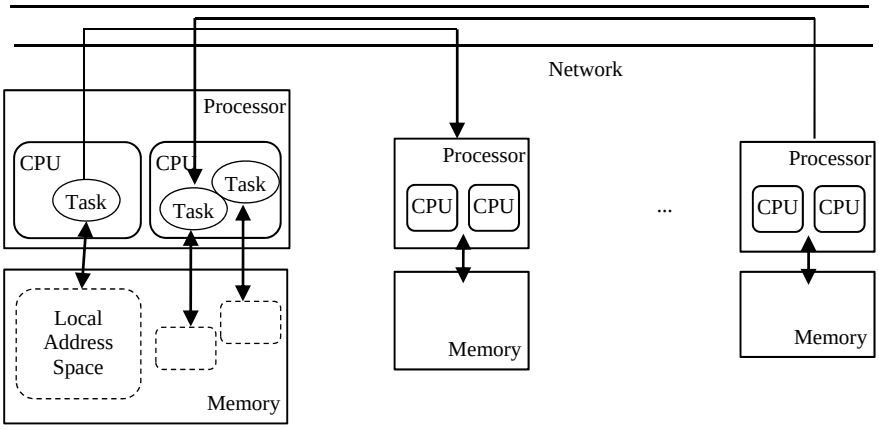

Figure 7. Architecture of distributed memory model implementation

Accessing memory nearer to a CPU is always faster than accessing remote or far away memory in computer systems. The per task memory abstraction thus naturally overlays with this property in hardware, thereby giving the benefit of improved performance through data locality to the programs. The main disadvantage of this abstraction is the local (fragmented) view of computing and the need to express communication explicitly. Section 3.1 presents the local view of computing in detail.

2.2.1 Hybrid Approach

It has been common to combine shared memory model implementations such as OpenMP or Pthreads with a distributed memory model implementation such as MPI for inter-core parallelism while keeping MPI as the mean for inter-node parallelism. The view of a program with this hybrid approach is shown in Figure 8.

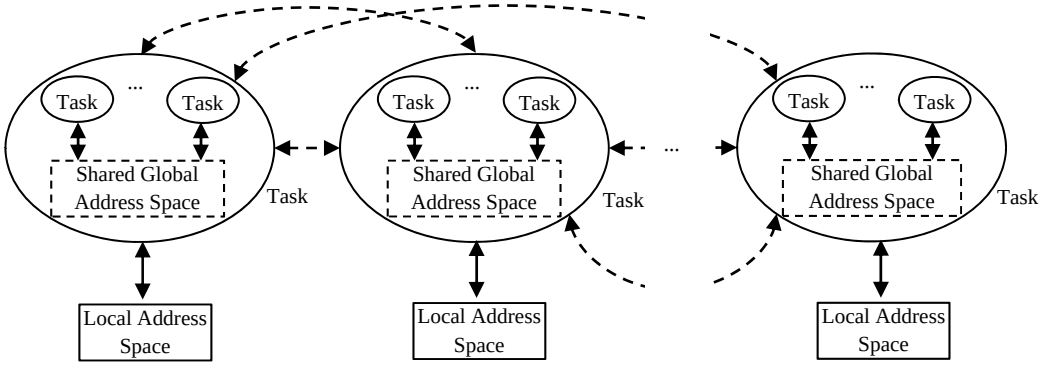

Figure 8. Programmer's view of the hybrid approach

There are two types of tasks visible in Figure 8, i.e. inter-node tasks and inter-core tasks, which are represented in large ellipses and small ellipses respectively. In practice, these may be MPI processes and Pthread or OpenMP threads. Each inter-node task is assigned with a local address space as in Figure 6, yet a portion of this is used by inter-core tasks as the shared address space among them. Total parallelism achievable in this setting is equal to the product of inter-node tasks and inter-core tasks. Also, in a multi-core environment where cores share physical memory, this approach seem to work efficiently than having pure MPI processes equal in parallelism.

2.2.2 Message Passing Interface (MPI)

MPI is an API specification allowing inter-process communication via message passing. Over the years it has become a *de facto* standard in communication among processes and is used by programmers commonly in realizing the distributed memory model. Implementations of the MPI specification are available for several languages like C, C++, and Fortran. Language bindings are available for some other high-level languages like Perl, Python, Ruby, and Java. Also two implementations are available for .NET based languages, i.e. Pure Mpi.NET and MPI.NET.

The execution of an MPI program requires specifying the total number of processes ahead of time. This number indicates the total parallelism of the solution and is fixed throughout the execution of the program. This constraint of fixed number of processes is relaxed in the set of MPI-2 extensions introduced over the original MPI-1 specification allowing an application to create and terminate processes after being started.

The typical execution model of an MPI program with a fixed number of N processes is depicted in Figure 9 where each MPI process is ranked from 0 to $N-1$.

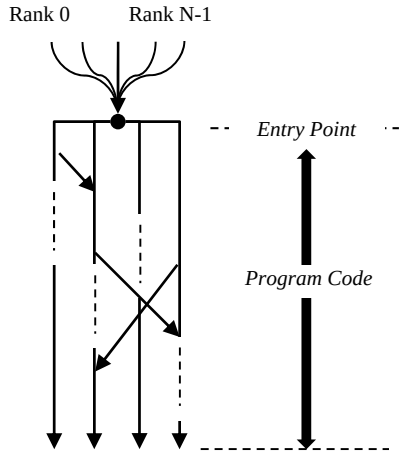

Figure 9: Execution model of an MPI program

Although not restricted, the usual practice with MPI is to execute the same program code by all the processes resembling the SPMD model of execution. The flow of execution, however, may differ across processes as instructed in the program code based on the rank. This is illustrated in Figure 9 by dashed lines in the flow of execution. Also, these processes operate on separate data entities since there is no notion of distributed data structures. Thus, data sharing is made explicit through

communication between processes as indicated by non-vertical arrows. MPI also supports patterns such as two-sided send and receive, broadcasts, reductions, and all-to-all.

MPI, though supports a fragmented view of computing, has been successful greatly as a mean of implementing parallel programs. MPI's success is credited for its six properties, i.e. portability, performance, simplicity and symmetry, modularity, composability, and completeness [7]. It is fair to mention that portability, performance, and completeness have attracted the scientific community towards MPI though other properties are equally important. The three properties are briefly discussed below.

Typical lifespan of parallel applications exceeds that of the hardware since high performance computing (HPC) systems tend to evolve rapidly. Thus, having an API like MPI, which does not require the knowledge of the target platform, makes programs easily portable across different hardware architectures.

The natural match of MPI's programming model and memory locality yields good performance values for MPI programs. Also, MPI being a library makes it possible to get the advantage of the best compilers available. MPI, however, is not able to give the best performance for a single source across different target platforms even though it supports both portability and performance. This lacking feature is defined as performance portability and realistically it is an unsolved problem even for Fortran on uniprocessor systems.

Finally, MPI is a complete programming model meaning that any parallel algorithm can be implemented with MPI, though it may not be easy. It has 128 routines in MPI-1 and 194 routines in MPI-2, to support such a general case of parallel programming problems. Unfortunately though, this rich collection of routines is often mistaken as an unnecessary complexity of the framework.

2.3 Partitioned Global Address Space (PGAS) Model

PGAS model tries to incorporate the best of both shared and distributed memory models, namely, simple data referencing and programmability of the former and data locality of the latter. It does so by providing a global address space, which is partitioned to make the tasks acting upon it aware of the locality. The programmer's view of the program with this model resembles a superimposed view of Figure 2 and Figure 6 as presented in Figure 10.

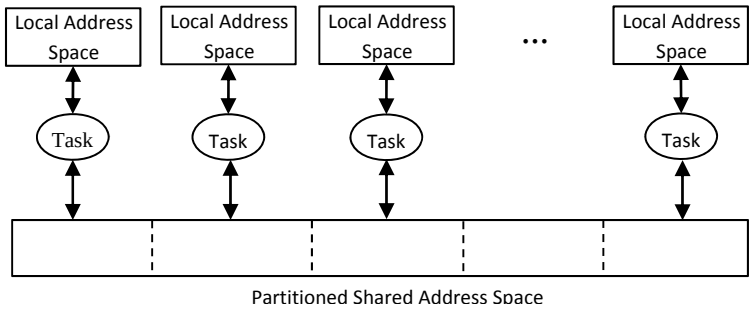

Figure 10. Programmer's view of a PGAS model

Each task has a private address space similar to the case with distributed memory model. Also, a portion of the total address space allocated for a task contributes to the shared space among tasks as well. The shared space in this model, however, enables a programmer to get the advantage of data affinity since there is a notion of partitions as shown by dashed vertical lines in Figure 9. This is different from the address space of shared memory model in Figure 2 where there was no ownership for global variables. Also, the presence of a shared address space removes the explicit communication exhibited in Figure 6 and Figure 8 for sharing data. Another importance difference, though not visible from Figure 10, is how implementations support the separation of shared and local variables. Unlike the usual lexically global shared-ness in shared memory model implementations, the PGAS languages explicitly differentiate global and local variables statically. For example, the variable allocation of a simple case is shown in Figure 11.

The array in Figure 11 brings out another feature that was not present with the distributed memory model, i.e. distributed data structures. The distribution of elements in the array may take either a round-robin, block-cyclic or other custom form supported by the implementation. In general, languages with a PGAS memory model may extend the support for distributed data structures beyond arrays as presented in a latter part of this paper.

Mapping of the PGAS model on to HPC hardware may be achieved with architecture similar to Figure 7 except the implementation is required to introduce necessary communication to refer remote memory. This is an advantage over the case

where communication is made explicit by the programmer. Thus PGAS languages can support a cleaner syntax for remote data access.

PGAS, similar to distributed memory model, starts off a program with a fixed number of execution units or threads. These threads work independently executing the program in an SMPD fashion. The model thus follows a local view as explained in section 3.1, though the presence of distributed data structures greatly reduces burden of data access. Also, the threads may exploit the locality of shared data to achieve better performance.

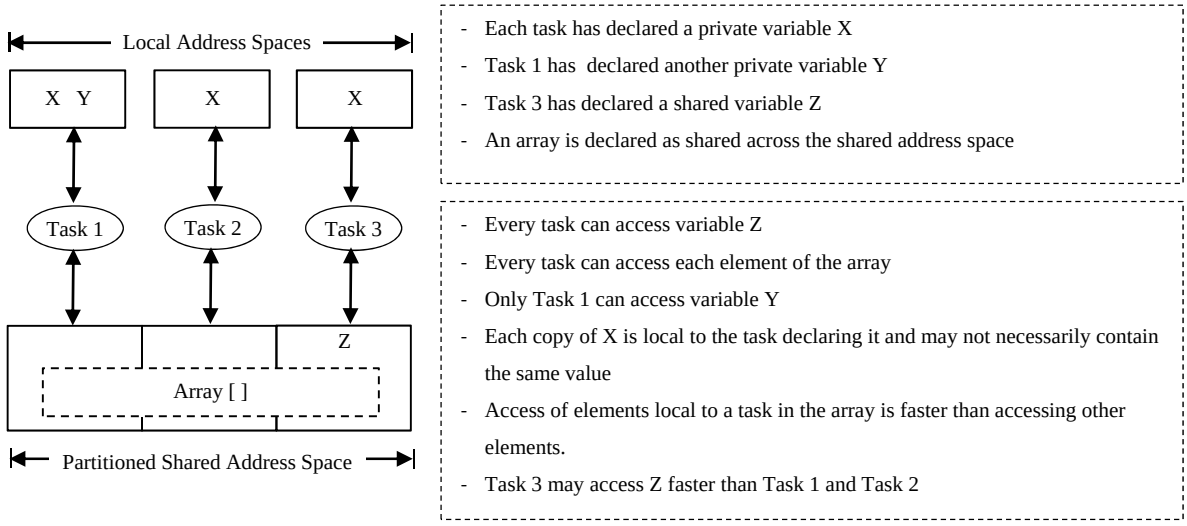

Figure 11. Variable allocation of a sample PGAS program

3. Views of Computing

The view of computing is another aspect of a language, which classifies current parallel computing support into two broad categories, i.e. local vs. global views of computing. Also note it is possible for a language to support both these views.

3.1 Local View of Computing

In Figure 1 we showed how one could decompose the sequential computation into orthogonal tasks, which get assigned to ACUs. It implies the decomposition is based on the properties of the algorithm but rather on the number of ACUs. However, some programming models may obligate the user to decompose the computation per ACU basis and to maintain the consistency through the orchestration phase. Models of this nature fall into the *fragmented* or *local* view of computing and SPMD is a well-known such programming model.

As an example, consider taking the three-point stencil of a vector [3]. The idea is to compute the average of left and right neighbors of each element in the array excluding the first and the last elements. The Figure 12 depicts a case where the original problem is decomposed into three tasks.

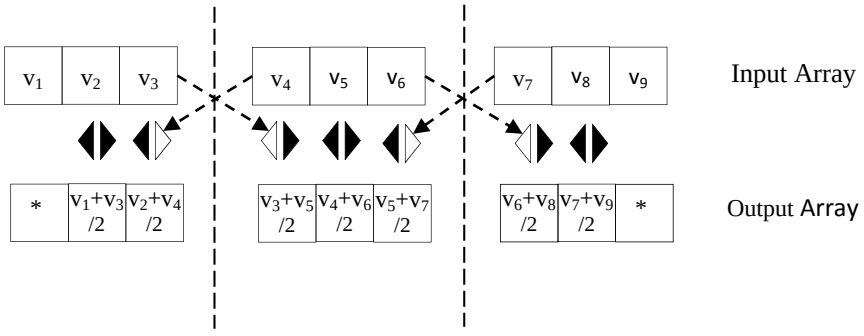

Figure 12: Local view of a 3-point stencil computation

The local availability of a neighboring value needed for the computation is denoted by a colored arrow head pointing to the direction of the neighbor. An uncolored arrowhead denotes the neighboring value on the directed side is not locally available. The program thus needs to incorporate additional logic to circumvent missing values by either explicit or implicit communication between partitions. This is denoted by dashed arrows pointing towards uncolored arrowheads in Figure 12.

Local view decompositions may also result in programs that tend to clutter the actual algorithm with details on communication and synchronization. Despite the loss of clarity in the solution and the possible burden for extra coding, presenting a local view may simplify the implementation of the language since each activity executes locally within a particular partition.

3.2 Global View of Computing

In global view computing task decomposition depends on the nature of the algorithm and they may not be fixed in number. In general, this leads to a more natural way of thinking and coding a solution than with a fragmented view. For example, the same three-point stencil may be decomposed into tasks that operate on a global array shown in Figure 13.

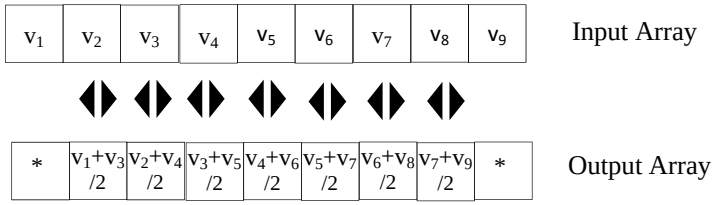

Figure 13: Gloabl view of a 3-point stencil computation

The code to the above solution will also become simple without the jargon of sharing intermediate values. A pseudo code is given below in Figure 14. Note that providing a global view may actually require partitioning and communication through the underlying implementation though not visible to the programmer.

```
Integer N = 9
Integer Array A[N], B[N]
Forall i = 2 to N – 1
       B[i] = (A[i-1]+A[i+1])/2
```

Figure 14: Pseudo code for global view 3-point stencil computation

It is worth examining how to deal with dependences of tasks under global view. For example if the updates happened to the same array in the above example it would result a chain of dependences among tasks to compute the intended solution. Note the intended solution is what one could expect from the serial version of the program. We would not be able perform parallel computations for this version of the 3-point stencil application. However, we may still wish to multiple tasks to utilize data locality when the array is distributed and those tasks may be synchronized using a global event array indicating when an element is ready for the computation of next element. In fact, a two dimensional 4-point stencil operation could benefit from such a method and perform some elements in parallel as described in [10] .

4. High Productivity Computing Systems (HPCS) Languages

Defense Advanced Research Project Agency (DARPA) recognized that difficulties in creating software systems to utilize the power of new architectures as a significant barrier in applying computations to science, engineering, and any other large scale data processing [11]. The High Productivity Computing Systems (HPCS) project followed up in 2002 with the intention of improving performance and usability of large scale parallel systems. Cray, IBM, and Sun competed in achieving DARPA's goals both in hardware and language design, the latter of which came in Phase II of the HPCS project as HPCS languages.

The languages include Chapel from Cray, X10 from IBM, and Fortress from Sun. These were initially targeted towards corresponding vendor's advanced computer systems, yet yielding comparable performance when run in commodity clusters.

4.1 Chapel

Chapel stands for Cascade High Productivity Language, where Cascade is the high productivity computer system that Cray is developing under HPCS project [14]. It is a new language not based on existing sequential languages since Chapel

developers believe looking too similar to an existing language may cause users to inadvertently write sequential style code rather considering their algorithms afresh [3].

It supports standard built-in floating point, integer, complex, boolean, and string types. Variable declaration syntax takes the form of `var <name> [: <definition>] [= <initializer>];` which differs from the common "type first" style. It also has structured, i.e. class, record, union, and tuple types. Other types in Chapel include enumerated, data parallel, and synchronization types.

Chapel has control flow support using loops, conditionals, select statements, breaks, continues, gotos, and returns as in standard block imperative languages. Similarly, it let users define functions with default argument values, argument matching by name, and argument intents. Chapel also includes object orientation, generic programming, locality control, and parallelism. The latter two are discussed in detail in sections 5.2.1, 5.4.1 and 5.5.1.

4.2 X10

X10 developers envision it to be a high productivity language aimed towards non uniform cluster computing while preserving the qualities of an object oriented language. Present day object oriented programming languages, such as Java and C#, have been widely adopted and are equipped with advance features such as managed runtime environments, virtual machines, automatic memory management, and concurrent programming [4]. X10 takes advantage of these by having its serial core based on a subset of Java. Its parallel core is based on PGAS model and is geared to support beyond uniprocessor systems.

X10 is a strongly typed language with limited local type inference. Its types are `class`, `struct`, `interface`, and `function` where the former three are categorized as unit types. Additionally, it supports constrained, annotated, and generic types. An example of a constrained type is `Int{self != 0, self != 1}`, which is the type of integers that are neither zero nor one. The constraint is a Boolean expression with a limited form [13].

It too supports control flow constructs such as loops, if statements, returns, throw, try, and catch as part of its serial core. Its parallel support is achieved through execution units called activities running on multiple places. A notion of clocks is introduced to synchronize activities. Detail information on these follow in sections 5.2.2, 5.4.2, and 5.5.2.

4.3 Fortress

The name "Fortress" comes from its intention to provide a "secure Fortran" for high performance computing with abstractions and type safety comparable to modern programming languages. Note, however, the language has no significance relationship to Fortran except the intended domain of application. Object oriented and functional languages have influenced Fortress design, but its syntax is not based on any of them [1]. In fact, the syntax is made as close as possible to mathematical notation since the developers believe scientists understand mathematical notation irrespective of their computing background [14]. For an example, juxtaposition of two variables as in `a = b c` indicates the multiplication of `b` and `c` without requiring the symbol '*'. Fortress also supports Unicode characters to be used and provides a code converter called Fortify that converts Fortress code into LaTex.

Fortress is a statically typed language, but supports type inference. The types in Fortress are trait, object expression, tuple, arrow, function, `Any`, `BottomType`, and `()`, where last three are special types [1]. Traits and objects are the building blocks of Fortress, where traits are similar to Java interfaces with the option of having concrete methods. Fortress is also an expression oriented language where everything is an expression and evaluates to a value. In cases such as assignment the type and value the expression evaluates to is `()`.

Parallel units of execution in Fortress are threads and they run on regions, which abstract the target architecture. Fortress provides atomic code blocks in order to synchronize data access with threads. Details on parallel support with Fortress are presented in sections 5.2.3, 5.4.3, and 5.6.3.

4.4 Common Features

Chapel, X10, and Fortress, irrespective of being implemented by different vendors, are expected to have four common qualities, i.e. creating parallelism, communication and data sharing, locality, and synchronization as identified in [11].

4.4.1 Creating Parallelism

HPCS languages include parallel semantics and one may exploit parallelism in forms of data parallel, task parallel, or nested parallel code. A significant difference in parallel semantics between HPCS languages and existing PGAS languages or libraries like MPI is the use of dynamic parallelism, where threads can be spawned inside application code at runtime. Other languages or libraries generally support static parallelism where parallelism is constant since start of the program and is mapped to a fixed process model. Dynamic parallelism benefits programmers in expressing as much parallelism as possible in their code, while the compiler and runtime determining the actual parallelism.

4.4.2 Communication and Data Sharing

Libraries like MPI use explicit message passing to communicate and share data between processes. HPCS languages, in contrast, use global address space for the same purposes. This has also made it possible for them to support distributed data structures such as distributed arrays. Global address is spaced advantageous in supporting collective operations, which are frequently used in scientific applications. In general it would require all processes invoking the global operation, but with HPCS languages one thread may perform the global operation on the shared data structure.

4.4.3 Locality

Data locality is important for performance reasons and all three HPCS languages provide means for the programmer to control the location of execution of threads. Chapel introduces the notion of locales to indicate abstract memory partitions. X10's notion of places is intended for the same purpose. Fortress's notion of regions is similar, but a region maps to a particular entity in computing structure such as node, processor, core, or memory, thereby making regions to form a tree.

4.4.4 Synchronization

Synchronization mechanisms are a necessity in general with parallel programs. Locks and barriers have been the common mechanisms in current parallel applications. HPCS languages, being based on dynamic parallelism, make barriers impractical. Locks are possible, but generally considered error prone and complicated to work with. Therefore, HPCS languages support atomic blocks as means to synchronize threads. Details on atomic blocks introduced in each language are present in 5.6. These guarantee the execution to be atomic with respect to all other operations in the program.

5. Idioms of Parallel Computing

Parallel computing often requires constructs for a variety of common tasks. Thus, programming languages and libraries usually come with a bunch of built-in constructs, which may be used alone or in conjunction with others to fulfill the desired objective. The usage patterns of these constructs are referred to as idioms henceforth and we present five idioms of parallel computing supported by each of the three HPCS languages in the following sections.

5.1 Taxonomy of Idioms

The following table presents the five common tasks identified in parallel computing and the corresponding idioms supported by the three HPCS languages - X10, Chapel, and Fortress. The idioms for Chapel were originally proposed in [12] and here we extend them to X10 and Fortress forming the basis of our comparison.

Table 1: Taxonomy of idioms

Language / Common Task	**Chapel**	**X10**	**Fortress**
Data parallel computation	`forall`	`finish … for … async`	`for`
Data distribution	`dmapped`	`DistArray`	`arrays, vectors, matrices`
Asynchronous Remote Tasks	`on … begin`	`at … async`	`spawn … at`
Nested parallelism	`cobegin … forall`	`for … async`	`for … spawn`
Remote transactions	`on … atomic`	`at … atomic`	`at … atomic`

The string literals in Table 1 represent the built-in constructs in each language and the ellipses indicate the combined use of two or more constructs in an idiom. A side-by-side comparison of each of these idioms is presented in the following sections.

5.2 Data Parallel Computation

Consider a constant `C` and two vectors `A`, `B` each with N elements participating in the following vector operation expressed in a simple loop construct.

```
for (i = 1 to N)
    A[i] = A[i] + C * B[i]
```

Figure 15. Simple vector operation

The computation in Figure 15 can be run concurrently for each data element, i.e. value of variable `i`. Thus the idea of data parallel computation is to perform concurrent evaluation of expressions relating to each data element. The language or library support for data parallel computation is the presence of convenient constructs that can yield such concurrent evaluation.

5.2.1 Chapel: `forall`

Chapel provides a convenient `forall` loop construct, which essentially evaluates the loop body for each data element or tuple in the iterable data structure. There are several long and short forms of this construct of which few examples are given in Figure 16.

Figure 16 a) shows how the `forall` construct may be used to iterate over multiple iterators simultaneously, which in Chapel's context is called a zippered iteration. The iterators are invoked simultaneously for their next value before each round of loop's body evaluation. The values returned by iterators are placed in order in the tuple, which acts as the data element of the iterator expression. Thus, in Figure 16 a) the tuple `(a,b,c)` is the data element of the iterator expression `(A,B,C)`, which in this case encapsulate three arrays. The simultaneous invocation of iterators guarantees that the $\texttt{i}^{th}$ element of each iterator is visible to the loop's body in $\texttt{i}^{th}$ iteration.

Figure 16 b) shows a variant of the `forall` construct, which iterates over an arithmetic domain instead of over elements of an array as used generally. Chapel's creators denote this kind of computation as data parallel computation without data since a domain in Chapel represent only a first class index set. An interesting feature of Chapel's domains is that their index sets may be distributed across multiple locales giving rise to distributed data structures. A locale is the abstract unit of target architecture presented by Chapel to its programs. More on data distribution is presented in section 5.3.

The `forall` construct is versatile in the sense that the same syntax could be used to iterate over a variety of array types such as local, distributed, distributed with different distributions, associative, sparse, or unstructured arrays. Moreover, it works equally well with user defined iterators thus giving the programmer the flexibility to iterate over data structures like trees and graphs, which are not necessarily arrays.

A loop constructed with `forall` is also a declaration by the user that the iterations of the loop may be evaluated concurrently. The degree of concurrency is left to Chapel's compiler, which may use one concurrent task per each invocation of the loop's body or use a less number of tasks thereby leaving one task to handle more than one invocation. The decision is based on the iterator statement, which determines the number of tasks to use, the number of iterations each task should handle, and the locale on which these tasks should execute. A strict variant, possibly with bit more overhead, of this model, which guarantees that each invocation of the loop's body is handled by a separate task, is Chapel's `coforall` form. Both forms, however, guarantee that the control is resumed with the statement following the loop only when all concurrent tasks evaluating loop's body are completed.

```
forall (a,b,c) in zip (A,B,C) do
    a = b + alpha * c;
```

a) Zipper iteration where multiple iterators are iterated simultaneously

```
forall i in 1 … N do
    a(i) = b(i);
```

b) Iteration over an arithmetic domain with an index range of [1 … N]

```
A = B + alpha * C;
```

c) A concise format of the particular computation in a)

```
[i in 1 … N] a(i) = b(i);
```

d) A concise format of c)

Figure 16. Forms of Chapel's `forall` construct

Chapel has a `forall` form that is usable in an expression context as well. It is very much similar to the `forall` statement form except returning an iterator of values, which resulted from each evaluation of loop's body. An example usage showing a reduction operation applied to such a list of values is shown in Figure 17, where the loop simply returns square of numbers from 1 to 10 and the reduction operation simply adds them together.

```
writeln(+ reduce [i in 1 .. 10] i**2;)
```

Figure 17. Chapel's `forall` in expression context

There are few implicit idioms of data parallel computation exist in Chapel other than the uses of explicit forms of `forall` such as whole array assignment, promotion, reductions, and scans. These, however, are not discussed within the scope of this paper.

5.2.2 X10: `finish … for … async`

X10 comes with a sequential `for` construct with the usual semantics of evaluating the loop's body for each data element in the iterable data structure. However, as opposed to Chapel, X10 does not come pre-equipped with a concurrent form to perform data parallel computations. Instead, it relies on the use of its asynchronous activity creation [section 5.4.2] together with the sequential `for` construct. The following shows two forms of X10's sequential `for` construct, resembling similar syntax to Chapel's `forall`.

```
for (p in A)
   A(p) = 2 * A(p);
```

a) Iteration over points in an array

```
for ([i] in 1 .. N)
   sum += i;
```

b) Iteration over numbers from 1 to 10

Figure 18. Forms of X10's `for` construct

An array in X10 is essentially a set of n-dimensional points each mapped to a value of given type. The set of points collectively identifies the region and acts as the index set of the array. Given a point `p` in array `A`, `A(p)` denotes the value associated with that particular point. The points of an array is iterable using the `for` loop construct as shown in Figure 18 a). A key difference between this and Figure 16 a) is that in X10 the iteration happens over the index set or set of points whereas in Chapel it happens over the elements of the array. Therefore, to access the array elements in X10's loop one needs to use the subscript notation, `A(p)`, as in Figure 18 a). Another difference is that X10 does not provide a zipper iteration form of the `for` loop that is capable of iterating over multiple arrays.

Figure 18 b) iterating over a set of single dimensional points while the coordinate value being assigned to the variable `i` as the loop index. The internals of this behavior is such that the `1 .. N` construct creates a set of `N` single dimensional points and the `[i]` destructs the point being picked for the iteration and assigns its coordinate to the variable `i`.

Arrays in X10 support distributions [section 5.3.2], which allow their elements to be spread across multiple locales – the abstract units of target architecture in X10. These distributed arrays work equally well with the `for` construct as local arrays do. X10 also supports iterations over user defined data structures implementing the `Iterable[T]` interface where `T` denotes the iterable type.

The sequential forms of the `for` construct may be coupled with the `async` construct to achieve concurrent evaluation of the loop's body as shown in Figure 19.

```
finish for (p in A)
   async A(p) = 2 * A(p);
```

Figure 19. X10's for combined with `async`

The `async` keyword instructs the X10 runtime to spawn a new activity to evaluate the statement following it [section 5.4.2] and immediately returns the control back to the statement following `async` block. Note the use of additional finish keyword in Figure 19, which makes the control to pass beyond the loop only when all the spawned activities are terminated. The

absence of the `finish` keyword will not produce an invalid X10 program, but may cause undesirable effects in the expected data parallelism since the control may pass the loop while the iterations are still active in spawned activities.

Also, note X10 supports two more variations of loops, i.e. `while` and `do-while` loops. These may also be used to iterate a collection like an array in parallel similar to the case of a `for` loop.

5.2.3 *Fortress:* `for`

Fortress provides a loop construct, i.e. `for`, which may run either sequentially or in parallel depending on the type of generator used. A generator is the one that produces elements of iteration. The four generators depicted in Figure 20 in order are, number range, indices of an array, elements of an array, and elements of a set. The bodies of the loops are evaluated in parallel as these generators are non-sequential. In fact, arrays, sets, lists, and maps are known as aggregate expressions in Fortress and are parallel generators by default. The implicit threads in the loop run in fork-join model, thus passing the control only when iterations complete.

Figure 21 illustrates how sequential iteration is possible even when using a non-sequential using the construct `sequential` to wrap the generator. Note elements in a set are indexed in their sorted order. Thus, sequential iteration in Figure 21 b) will result the sequence `1, 3, 6, 8, 10` instead of `1, 3, 10, 8, 6`.

```
for i <- 1:10 do
       A[i] := i end
```

a) Iteration over a number range

```
A:ZZ32[3,3]=[1 2 3;4 5 6;7 8 9]

for (i,j) <- A.indices() do
       A[i,j] := i end
```

b) Iteration over the indices of an array

```
for a <- A do
       println(a) end
```

c) Iteration over the elements of an array

```
for a <- {[\ZZ32\] 1,3,5,7,9} do
  println(a) end
```

b) Iteration over the elements of a set

Figure 20. Forms of Fortress's `for` construct

```
for i <- sequential(1:10) do
       A[i] := i end
```

a) Sequential iteration over a number range

```
for a <- sequential({[\ZZ32\] 1,3,10,8,6}) do
       println(a) end
```

b) Sequential iteration over the elements of a set

Figure 21. Sequential iteration in Fortress

There are few other types of implicit data parallel construct exist in Fortress such as tuple expressions, `also do` blocks, arguments in a function call, sums, guards, and tests. Similar to the `for` loop these are run with implicitly spawned threads and a Fortress implementation may even serialize portions of any group of implicit threads.

5.3 Data Distribution

A parallel program typically runs on a collection of separate nodes utilizing their physically distributed memories, which implies that irrespective of the memory abstraction the memory access times are bound to be non-uniform. So it is

advantageous to distribute data structures like arrays across nodes when possible to achieve data locality. The distribution, however, is expected not to change the semantics of the rest of the program.

5.3.1 Chapel: `dmapped`

The intrinsic data structure to hold a collection of same type values in Chapel is an array. Arrays in general programming languages are accessed using index values which range usually from zero to one less than the total size of the array. Also, such index values are linear consecutive integers and are integral part of the array itself. Chapel deviates from this usual approach by separating the index set from the data buckets that are mapped by the index set. The separated first class representation of an index set is defined as a domain in Chapel, which in turn is used to define arrays. Moreover, domains may be dense, stridden, sparse, unstructured, or associative. A simple example is given in the Figure 22.

```
var D: domain(2) = [1 .. m, 1 .. n];
var A: [D] real;
```

Figure 22. Chapel's domain and array example

First line of Figure 22 defines a two dimensional domain where the index ranges are from `1` to `m` and `1` to `n`. Note, variables `m` and `n` may be specified at complied time or get evaluated to values at runtime. The second line in the example creates an uninitialized array of real values out of the particular domain.

Domains and arrays present a global view to the programmer, yet they need to be mapped into the multiple locales [section 5.4.1] giving rise to data distribution. The mapping is called a domain map and it defines how to map domain indices and array elements into different locales as well how to layout them in memory within a locale. The construct `dmapped` is used to specify the domain map for a domain as shown in Figure 23.

```
const D = [1..n, 1..n];
const BD = D dmapped Block(boundingBox=D);
var BA: [BD] real;
```

Figure 23. Chapel's `dmapped` construct

Line one in Figure 23 creates a two dimensional domain `D`, which in the absence of explicit mapping information will get mapped using the default layout to the current locale. Line two creates a new domain, `BD`, based on `D` with an explicit block distribution specified using the `dmapped` construct. The `boundingBox` parameter specifies which index should be considered for distribution, which in this case is as same as the domain `D`. Last line shows how to create an array from the domain `BD`. Elements of array `BA` will get distributed in the same way as the domain that was used to create it.

Chapel comes built-in with a set of domain maps and also provides the flexibility to implement user defined mappings. Interestingly, both the built-in and any user defined domain maps are written purely using Chapel itself. More on Chapel domain maps can be found their language specification in [9].

5.3.2 X10: `DistArray`

X10 has taken a similar approach to Chapel when it comes to arrays, i.e. separate index set from actual data buckets. The notion of a point is introduced in X10 to represent an index of any dimension. The dimensionality of a point is called the rank and a collection of points with the same rank defines a region in X10. Similar to domains in Chapel, regions in X10 in turn define arrays. An example on creating regions and arrays is given in Figure 24.

```
val R = (0..5) * (1..3);
val arr = new Array[Int](R,10);
```

Figure 24. X10's region and array example

First line of the example creates a rectangular region consisting of two dimensional points, ordered lexicographically as `(0,1),(0,2),(0,3)` … `(5,1),(5,2),(5,3)`. The second line uses this region and creates an integer array, while initializing all elements to number 10.

Points of a region may be distributed across multiple places using X10's notion of a distribution. An array can be constructed over a distributed set of points using the construct `DistArray` in X10, which will distribute the data buckets holding the array elements according a given distribution of points. An example is given in Figure 25.

```
val blk = Dist.makeBlock((1..9)*(1..9));
val data : DistArray[Int]= DistArray.make[Int](blk, ([i,j]:Point(2)) => i*j);
```

Figure 25. X10 distributions and distributed arrays

The top line of the example creates a square region and maps points into places using a block distribution. The standard block distribution implementation tries to distribute points as evenly as possible among multiple places in X10. The next line shows how to create a distributed array using the block distribution and also to initialize each element using an anonymous function.

X10's distribution class, `Dist`, has built-in support for several distribution types like block, block block, cyclic, block cyclic, unique, and constant. This class may be extended to support user-defined distributions as well.

5.3.3 Fortress: Distributed Arrays, Vectors, and Matrices

Fortress is intended to support data distribution through its array, vector, and matrix data structures. Data distribution is expected to be done by default without requiring explicit constructs in contrast to Chapel and X10. Fortress specification mentions several distributions such as `blocked`, `blockCyclic`, `columnMajor`, `rowMajor`, and `default` [8]. Similar to the distributions available in Chapel and X10 these decide the placement of data, which is associated with each data element as its region in Fortress region hierarchy [section 5.4.3]. Moreover, these are meant to be implemented as user level libraries written purely in Fortress language itself. In its current state, however, Fortress does not have working implementations for distributions [1].

5.4 Asynchronous Remote Tasks

Shared memory implementations like OpenMP and Pthreads provide the capability to spawn tasks dynamically within the code in contrast to fixed process model of distributed memory implementations like MPI or PGAS implementations like CAF, UPC, and Titanium. Shared memory implementations being restricted to execute as a single process on a single node, however, make dynamic task creation unusable across processes – presumably running on multiple nodes. Hence, the idea of asynchronous remote tasks is to enable a task running in one location to spawn tasks asynchronously in another location. The term location refers to the abstract unit of target architecture presented to the program by the underlying language. Asynchrony implies that the creator task does not have to wait till spawned tasks complete to resume its work.

The dependencies among tasks (orchestration) need to be handled through global data structures and synchronization constructs. These synchronization constructs guarantee some form of ordering of events. Therefore, the following sub sections briefly touch into synchronization mechanisms available in each of the three languages as well. Also note there is no mechanism to pass messages between ACUs.

5.4.1 Chapel: `on … begin`

In the context of Chapel, a locale is the abstract unit of target architecture and a task is the abstract unit of computation. Most of the traditional parallel programming tools, in contrast, present a single abstraction such as a process for both the unit of target architecture and computation, which hinders the idea of remote tasks.

The construct `begin` creates a new task to evaluate the statement following it in the same locale as the calling task. An example is given in Figure 26.

```
begin writeline("Hello");
writeline("Hi");
```

Figure 26. Chapel's `begin` construct

The control returns immediately to the calling task of the `begin` construct. Thus, the output for the program shown in Figure 26 will be either `Hello` followed by `Hi` or vice versa. Note `writeline` is an indivisible operation.

The evaluation of a statement may be migrated to a different locale using the `on` construct, which has the form `on` *`expression`* `do` *`statement`* where the *`expression`* must evaluate to a locale. Thus, combining these two forms one could achieve asynchronous remote tasks with Chapel as shown in the code fragment of Figure 27.

```
on A[i] do begin
   A[i] = 2 * A[i]
writeline(“Hello”);
writeline(“Hi”);
```

Figure 27. Chapel's `on` combined with `begin`

Array `A` in the above code is assumed to be distributed across locales. The code will move the evaluation from current locale to the locale containing `i`th element of `A` and return immediately to the `writeline` statement after creating a task to handle the multiplication and assignment of `A[i]`. In general, given a function `f()` and a locale `loc` one can use the idiom `on loc do begin f()` to evaluate `f()` on `loc` asynchronously.

The construct `begin` is an unstructured form of introducing asynchronous tasks in Chapel and is followed by two unstructured synchronization constructs, i.e. `sync` and `single` variables. These are type qualifiers that change the semantics of reading and writing to variables. The `sync` qualifier will make variables to have a state either as *empty* or *full*. An assignment operation to such a variable will block if its state is *full* and will proceed only if it is changed to *empty* by a read operation. Similarly a read operation will block if the state is *empty* and will proceed only it is changed to *full* by an assignment operation. Reads and writes will atomically change the state of `sync` variables. The `single` variables are a specialization of `sync` variables such that they can be assigned a value only once during its lifetime. Read operations executed on a `single` variable prior to any write operations will cause them to block. Once a value is assigned a `single` variable acts as a regular variable.

Chapel also comes equipped with few structured forms to introduce asynchronous tasks and synchronization such as `cobegin`, `coforall`, `sync`, and `serial`. The `cobegin` is followed by a block of statements each of which is evaluated by a separate asynchronous task. Unlike the `begin` statement, however, `cobegin` does not return control until all the spawned tasks are completed. Note the spawned tasks may in turn spawn more tasks, but `cobegin` is not held till those complete. The `coforall` statement, as discussed in 5.2.2, is similar to `cobegin` except each evaluation of the loop body is handled by a separate task. The `sync` statement will wait till all the concurrent tasks to complete including those spawned by other tasks, before returning the control. Finally, the `serial` construct is followed by an expression, which evaluates to a Boolean, and a statement or a block of statements. If the Boolean expression is true then any explicit or implicit task creation will be suppressed. Essentially, the statements following the `serial` construct will get evaluated serially.

5.4.2 *X10:* `at … async`

X10 also provides an abstraction for the unit of target architecture called a place. A programmer may think of it as a repository for data and activities, loosely resembling the notion of a process. An activity is the abstract unit of computation in X10. All the activities in X10 form a tree rooted at the main activity, which is responsible for launching the X10 program.

The construct `async` spawns a new activity in the current place and has the basic form of `async` *`statement`*. For example consider the an activity `T` spawning two more activities, T1 and T2, asynchronously as show in Figure 28 to evaluate statements S1 and S2.

```
{                          // activity T
    async {S1;}            // spawns T1
    async {S2;}            // spawns T2
}
```

Figure 28. X10 `async` construct

The activity T in Figure 28 terminates after spawning T1 and T2, which is said to be a local termination. T1 and T2 may spawn other activities and terminate locally too. Activity T is said to terminate globally when T1, T2, and any descendant activities terminate locally.

The place where a statement will be evaluated is changed using the `at` construct in X10, which has the general form of `at (q) S` where `q` should evaluate a place. The X10 runtime identifies the objects reachable by `S` in its lexical scope and serializes them to the place of `q` at which they are recreated and the evaluation of `S` proceeds. The control returns back to the original place when S terminates and if errors occur they are serialized back to the original place as well.

The construct `at` used with `async` gives the capability of creating asynchronous remote tasks and each of the following combinations may be used as required.

- `at (p) async S`
  - migrates the computation to `p` and spawns a new activity in `p` to evaluate `S` and returns control
- `async at (p) S`
  - spawns a new activity in current place and returns control while the spawned activity migrates the computation to `p` and evaluates `S` there
- `async at (p) async S`
  - spawns a new activity in current place and returns control while the spawned activity migrates the computation to `p` and spawns another activity in `p` to evaluate `S` there

An example of such usage is shown in the code fragment in Figure 29 using the second combination.

```
finish for (p in unique.places()) async {
    at (p) {
        for ([i] in dist|here) {
            val x = (i + 0.5) * step;
        }
    }
}
```

Figure 29. X10's `at` combined with `async`

In the above example `unique` is a distribution [section 5.3.2] and `unique.places()` returns the set of places it is distributed over. Similarly `dist` is a distribution as well and `dist|here` returns the points in the distribution that are located in the current place. The variable `step` is a constant.

The construct `finish` as used in Figure 29 is one synchronization in X10, which will wait for the loop to terminate globally before passing the on to the evaluation of next line. X10 clocks are another form of synchronization available, which resembles barriers in usual parallel programming. The idea behind clocks is that parallel programs proceed in phases and often synchronization is required between phases as in a fork-join model. An activity may register with one or more clocks using a syntax `async clocked (c1,c2) S` where `c1` and `c2` are instances of `Clock` class. The activity may signal its completion of a phase to either clock by calling for example `c1.resume()`. If the activity wishes to wait for other activities that are registered on `c1` to indicate the completion of the particular phase, it may do so by calling `c1.next()`, which is a blocking operation. In particular it is advised to use the statement `next` instead of explicitly calling `next()` on each clock to avoid deadlocks.

5.4.3 Fortress: `spawn … at`

Fortress provides a hierarchical abstraction of the target architecture through its regions in contrast to the flat memory abstractions of Chapel and X10. This is depicted in Figure 30.

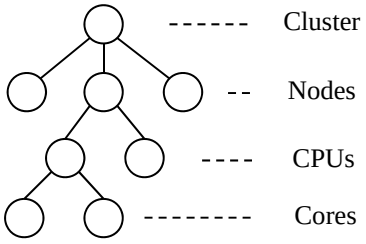

Figure 30. Fortress region hierarchy

All threads, objects, and array elements in Fortress have an associated region, which can be queried through the method `region()` on them. Objects near the leaves of the region hierarchy are the most local and those that reside at higher levels are logically spread out. However, the placement of an object in the hierarchy does not affect the semantics of the program, instead serves for performance purposes [1].

The unit of execution in Fortress is a thread and it can be placed near data using the `at` and `do` constructs as in Figure 31.

```
(v,w) := (exp1,
          at a.region(i) do exp2 end)
```

```
spawn at a.region(i) do exp end
```

```
do
       v := exp1
       at a.region(i) do
              w := exp2
       end
       x := v+w
end
```

Figure 31. Fortress's at...do constructs

The top code snippet of Figure 31 shows the creation of a tuple where the two elements are `v` and `w`. They are computed through expressions *exp1* and *exp2* respectively where the latter is placed in the region of `i`th element of array `a`. Note tuple construction implicitly spawns threads to evaluate its two expressions. The second code sample illustrates explicit spawning of a thread in a remote region. It produces a thread with type of *exp* that will run in parallel with succeeding code. The last sample depicts the ability to use `at … do` construct within a `do` block. A thread group is formed out of a single implicit thread evaluating the outer `do … end` block. It shifts its region when evaluating `exp2` inside `at … do`. The control returns back when evaluating `exp2` completes. This can be thought of as a limited case of `also … do` blocks in Fortress, which spawns threads for each block. The mechanism to synchronize spawned threads in Fortress is to use the atomic expressions, which are discussed in 5.6.3.

5.5 Nested Parallelism

Once data and task parallelism is possible it becomes interesting and useful to be able to use them conjunctively. As an example it may be useful to execute multiple data parallel statements in parallel. The reverse may also be useful in situations as discussed later.

5.5.1 Chapel: `cobegin … forall`

As explained in previous sections, the construct `cobegin` will spawn tasks to evaluate its statement list and `forall` tries to evaluate iterations of the loop body in parallel. Therefore, to evaluate multiple data parallel code in parallel, one could use a similar approach suggested in [12] and shown below in Figure 32.

```
cobegin {
       forall (a,b,c) in (A,B,C) do
              a = b + alpha * c;
       forall (d,e,f) in (D,E,F) do
              d = e + beta * f;
}
```

Figure 32. Nesting data parallelism inside task parallelism in Chapel

In the above example, the two `forall` loops are evaluated by parallel tasks and they in turn evaluate their iterations in parallel. The control pass beyond the `cobegin` statement only when both loops are done with iterations.

Another interesting scenario would be to spawn tasks inside data parallel code. A simple example is shown in Figure 33.

```
sync forall (a) in (A) do
              if (a % 5 ==0) then
                     begin f(a);
              else
                     a = g(a);
```

Figure 33. Nesting task parallelism inside data parallelism in Chapel

The example in Figure 33 calls a function `f()` for every fifth element. This operation is assumed to be time consuming and hence run on separate tasks. This would not make sense if each evaluation of the loop's body is run on a parallel task as in `coforall`, but remember with `forall` one task may evaluate more than one instance of the loop's body. Also, note the use of `sync` construct to make sure that when control passes to the next line any dynamically spawned tasks have completed.

5.5.2 X10: `for … async`

Any X10 code written run as data parallel may be evaluated in a dynamically spawned activity with the help of `async` construct. A simple template code for this is given in Figure 34.

```
finish { async S1; async S2; }
```

Figure 34. Nesting data parallelism inside task parallelism in X10

In the example it is assumed that `S1` and `S2` are data parallel code. Note the use of finish construct, which will delay passing the control till both activities terminate.

X10, however, does not directly support a data parallel loop as explained in section 5.2.2, yet as mentioned there one could achieve parallel iteration over a collection of data items. Thus, it is possible to form real code adhering to the pattern shown above.

Given a data parallel code in X10 it is possible to spawn new activities inside the body that gets evaluated in parallel. However, in the absence of a built-in data parallel construct, a scenario that requires such nesting may be custom implemented with constructs like `finish`, `for`, and `async` instead of first having to make data parallel code and embedding task parallelism.

5.5.3 Fortress: `for … spawn`

Data parallel code written in Fortress may be instructed run in parallel with any succeeding code by nesting it inside an implicit or explicit thread. First example of Figure 35 shows the use of an explicit thread to run a data parallel expression, *exp*. Note it might be necessary to wait on the thread for consistency, which can be done by calling the `wait()` method on the thread. The second example illustrates the use of structured parallel constructs in Fortress to nest data parallel expressions, *exp1* and *exp2*. Any code following the `do … end` block will resume only after both *exp1* and *exp2* are evaluated in parallel.

```
T:Thread[\Any\] = spawn do exp end
T.wait()
```

```
do exp1 also do exp2 end
```

Figure 35. Nesting data parallelism inside task parallelism in Fortress

The reverse, i.e. spawning threads within data parallel code, is also possible with Fortress using either explicit or implicit threads.

```
arr:Array[\ZZ32,ZZ32\]=array[\ZZ32\](4).fill(id)
for i <- arr.indices() do
      t = spawn do arr[i]:= factorial(i) end
      t.wait()
end
```

Figure 36: Nesting task parallelism inside data parallelism in Fortress

Figure 36 presents the use of explicit threads within the context of a parallel `for` loop where `i`th element is assigned its factorial value. In fact, such nesting would make more sense when there is just more than one thing to do with `i`th element.

5.6 Remote Transactions

The idea of remote transactions is to extend the general notion of atomic regions such that a task may declare a piece of code designated to run in a remote location as atomic.

5.6.1 *Chapel:* `on … atomic`

The Chapel development team is still implementing atomic statement support and has not decided whether to support strong or weak atomicity. The former will make an atomic statement to appear atomic from the point of view of all other tasks. The latter will make it so only with respect to other atomic statements.

One may make a statement atomic by prefixing it with the keyword `atomic` and unlike an implementation based on locks, atomic statements are composable. However, it is not be possible to make any arbitrary code atomic. For example, system calls may not be put inside an atomic block.

The pattern to make a remote atomic statement is to use the `on` construct as shown in Figure 37.

```
on A[i] do atomic A[i] = 2*i;
```

Figure 37. Remote transactions with Chapel

5.6.2 *X10:* `at … atomic`

Atomic blocks in X10 come in two flavors, i.e. unconditional and conditional, and provide means to coordinate access to shared data. The atomicity is weak in the sense that an atomic block appears atomic only to other atomic blocks running at the same place. Atomic code running at remote places or non-atomic code running at local or remote places may interfere with local atomic code, if care is not taken. Thus, as a rule of thumb X10 suggests that if any access to a variable is done in an atomic section then all access to that variable must be done in atomic sections [13]. Weak atomicity, however, permits an efficient implementation over global (absolute) atomicity.

Unconditional atomic blocks follow the simple form, `atomic S`, where atomicity is guaranteed if `S` terminates successfully without exceptions. Restrictions such apply on S such as it may not spawn other activities, not use blocking statements, not `force()` a `Future`, and not use `at` expressions. Unconditional atomic blocks has the equivalence property that `atomic {S1; atomic S2} is equal to atomic {S1, S2}`. Also, atomic blocks are guaranteed not to introduce deadlocks [13]. Figure 38 shows an example where an integer variable is incremented by two activities. The left code snippet displays the correct use of atomic blocks, which results the intended `n=3` at the end. The code on right is legal, but may cause unwanted interleaving of operations resulting `n=1` or `n=2` in addition to the correct `n=3`. Note, n=2 could not happen under absolute atomicity, but X10 preserves atomicity with respect to other atomic blocks only [13].

```
var n : Int = 0;
finish {
      async atomic n = n + 1; //(a)
      async atomic n = n + 2; //(b)
}
```

```
var n : Int = 0;
finish {
      async n = n + 1; //(a) -- BAD
      async atomic n = n + 2; //(b)
}
```

Figure 38. X10's unconditional atomic blocks

Conditional atomic blocks take the form of `when (B) S` where execution of `S` waits till Boolean condition `B` is true. Also, evaluation of B is atomic with S making it impossible to interrupt between `B` and `S`.

```
def pop() : T {
      var ret : T;
      when(size>0) {
            ret = list.removeAt(0);
            size --;
      }
      return ret;
}
```

Figure 39. X10's conditional atomic blocks

Figure 39 demonstrates the implementation of the pop operation of a list, where the removal of top element and the decrement of size are done in an atomic block that happens only when the list is not empty. A call to `pop()` will block if the list is empty and will get triggered to reevaluate the condition only if the `size` variable is changed within another atomic block, i.e. state changes occurring outside atomic blocks do not guarantee to trigger calls that are blocked on them to reevaluate the `when` conditions. Also, X10 does not make fairness or liveness guarantees.

X10's atomic blocks, both unconditional and conditional, may be used at remote places using the `at` construct.

```
val blk = Dist.makeBlock((1..1)*(1..1),0);
val data = DistArray.make[Int](blk, ([i,j]:Point(2)) => 0);
val pt : Point = [1,1];

finish for (pl in Place.places()) {
   async{
      val dataloc = blk(pt);
      if (dataloc != pl){
         Console.OUT.println("Point " + pt + " is in place " + dataloc);
         at (dataloc) atomic {
            data(pt) = data(pt) + 1;
         }
      }
      else {
         Console.OUT.println("Point " + pt + " is in place " + pl);
         atomic data(pt) = data(pt) + 2;
      }
   }
}
Console.OUT.println("Final value of point " + pt + " is " + data(pt));
```

Figure 40. Remote transactions in X10 with `at ... atomic` construct

Figure 40 illustrates the access to remote data inside an atomic block. The first three lines make a distributed array of a single element for the purpose of the example. This element is local to one place in X10 and when the above code is run with multiple places only the activity running in that particular place can access it without requiring a place shift. The intention of the code is to increment the array element atomically by activities running on multiple places. In particular, remote activities increment it by one, while the local activity increments it by two. Thus, remote activities use the pattern `at(P) atomic S` to perform the remote transaction, where `P` evaluates to the place containing the array element.

5.6.3 Fortress: `at … atomic`

Fortress provides the `atomic` construct to make code blocks run as transactions where all threads except the one running the atomic expression observe it as either completed or not started.

```
do
       x:Z32 := 0
       y:Z32 := 0
       z:Z32 := 0
       atomic do
              x += 1
              y += 1
       also atomic do
              z := x + y
       end
       z
end
```

Figure 41. Fortress's atomic expressions

Figure 41 illustrates a simple case of atomic expressions where one atomic region performs an update on variables and the other references the particular variables. Note these two blocks of code are run on two threads due to the presence of `also` construct. Possible outcomes for the variable, `z`, are 0 and 2, but not 1. In particular, it is possible for the thread executing the second atomic block to find an updated value for `x` but not for `y`. In this case it will be a read/write collision and on thread will abort. The aborted thread will rollback any updates made to variables and back off via spin and retry.

Atomic transactions may also be instructed to run in remote regions using the `at` construct as shown in Figure 42.

```
f(y:ZZ32):ZZ32=y y
D:Array[\ZZ32,ZZ32\]=array[\ZZ32\](4).fill(f)

q:ZZ32=0
at D.region(2) atomic do
      println("at D.region(2)")
      q:=D[2]
      println("q in first atomic: " q)
also at D.region(1) atomic do
      println("at D.region(1)")
      q+=1
      println("q in second atomic: " q)
end
println("Final q: " q)
```

```
at D.region(1)
at D.region(2)
q in second atomic: 1
at D.region(1)
q in second atomic: 1
at D.region(2)
q in first atomic: 4
Final q: 4
```

Figure 42. Remote transactions in Fortress with `at ... atomic` construct

In Figure 42, `D` is a single dimension array of integers and its elements are 0,1,2, and 4. The first and second atomic blocks are instructed to run on the regions where `D`'s third and first elements exist respectively. Possible outcomes for the final value of q are 4 and 5. The output for the case resulting value 4 is given on the right hand side of Figure 42 where it also illustrates a collision and spin/retry. Apparently, both threads have entered their atomic sections and thread two has detected a collision and has retried later.

6. Example Code

We have picked k-means clustering to demonstrate the use of above idioms in practice. The idea of k-means clustering is to partition *N* data points into *K* clusters by assigning points to the nearest cluster center. The program starts with a random *K* number of centers. These centers are then refined iteratively by taking the mean of nearest points to each center. The refinement continues until the change in centers falls below a predefined threshold. This algorithm is simple to comprehend, yet broad enough to exploit most or all of the above idioms

Chapel and X10 implementations of the algorithm support multiple locales and places respectively. Also, they follow a similar structure in the code. The Fortress implementation, however, does not use explicit distributions due to its lack of compiler support and documentation.

6.1 Chapel: K-Means Clustering

The following is k-means algorithm implemented in Chapel and is presented in segments highlighting only the details relevant to the idioms for clarity. Full working sample is presented in Appendix A.

- Declaration of points, dimension of a point, iterations, and number of clusters

```
config var numDim: int = 2,
           numClusters: int = 4,
           numPoints: int  = 2000,
           numIter: int = 50,
           threshold: real = 0.0001;
```

  Note. These are configurable at start time by passing them as command line arguments. For example passing `--numPoints 50000` will set the number of points to 50000 instead of 2000.

- Definitions of domains

```
const PointsSpace = {0..#numPoints,0..#numDim};
const ClusterSpace = {0..#numClusters,0..#numDim};
const ClusterNumSpace = {0..#numClusters};
```

  The `{0 .. #N}` notation indicates the integer range starting at zero and counting up to `N` number of values, i.e. `0, 1, 2 ... N-1`

- Block distribute points along zeroth dimension across locales.

```
var blockedLocaleView = {0..#numLocales,1..1};
```

```
var blockedLocales: [blockedLocaleView] locale = reshape(Locales, blockedLocaleView);
const BlockedPointsSpace = PointsSpace dmapped Block(boundingBox=PointsSpace,
                                                     targetLocales=blockedLocales);
var points: [BlockedPointsSpace] real;
```

The array `Locales` and the `numLocales` are made available to programs by the Chapel runtime and we have reshaped the locales into a two dimensional array of `numLocales x 1` size. This ensures components of each point stay in the same locale when blocked.

- Arrays to hold current centers.

```
var currCenters: [ClusterSpace] real;
currCenters = points[ClusterSpace];
```

Chapel's rich array operations make it possible to assign a subset of points as current centers by specifying a range as shown in second line.

- Replicated arrays to keep local centers and updates.

```
const ReplClusterSpace = ClusterSpace dmapped ReplicatedDist();
var localCurrCenters: [ReplClusterSpace] real;
var localCenterUpdates: [ReplClusterSpace] atomic real;
const ReplClusterNumSpace = ClusterNumSpace dmapped ReplicatedDist();
var localCenterPCounts: [ReplClusterNumSpace] atomic int;
```

Chapel allows transparent access to array elements on remote locales; however, performance may suffer when data is transferred over network. Therefore, it is beneficial to use local arrays when operating on points in a particular locale. Chapel provides a convenient distribution to achieve this, ReplicatedDist, which creates a copy of the array in each locale. The array reference resolves to the local copy in the particular locale where the referring code runs.
Also note the use of atomic real and integer variables, which is done as a workaround to non-implemented atomic blocks.

- The next steps happen iteratively while refining centers. We start by resetting local arrays as follows.

```
cobegin {
  // copy currCenters to each replicand
  localCurrCenters = currCenters;
  // reset local updates and counts
  forall lcu in localCenterUpdates do lcu.write(0.0);
  forall lcpc in localCenterPCounts do lcpc.write(0);
}
```

The second line copies the current centers to `localCurrCenters` array in each locale. The `ReplicatedDist` in Chapel guarantees the array assignment in the first line happens in each locale. The next two lines initialize the two local arrays, i.e. `localCenterUpdates` and `localCenterPCounts`, to zero. Again, the distribution guarantees the two forall loops happen for each local copy of the arrays. We run these three statements in parallel by wrapping them with a `cobegin` clause.

The next is to compare the distance for each point against all cluster centers and decide the cluster it belongs to. The code for this is as follows.

```
forall p in {0..#numPoints} {
  on points[p,0] {
    var closestC: int = -1;
    var closestDist: real = MaxDist;
    for c in {0..#numClusters} { // possibility to parallelize
      var dist: atomic real;
      dist.write(0.0);
      forall d in {0..#numDim} {
        var tmp = points[p,d] - localCurrCenters[c,d];
```

```
        dist.add(tmp*tmp);
      }

      if (dist.read() < closestDist) {
        closestDist = dist.read();
        closestC = c;
      }
    }

    forall d in {0..#numDim} {
      atomic { // would have been great, but not implemented yet in Chapel
        localCenterUpdates[closestC,d].add(points[p,d]);
      }
    }
    localCenterPCounts[closestC].add(1);
  }
}
```

Note the shifting of locales using the `on` clause in line 2, which in turn gurantee the access of current centers, center updates, and center point counts arrays local to the particular locale. We have placed the `atomic` construct to show where the updates should be done atomically, but it is not implemented in Chapel yet. However, the use of atomic variables, overcomes this and guarantees proper atomic updates. Also note if the number of clusters was large, the `for` loop could be changed to the parallel `forall` version with slight modification to the code. Moreover, if parallel task creation was unnecessarily expensive one could easily change the code to use serial execution.

Next we collate centers in each locale as follows.

```
var tmpLCU: [ClusterSpace] atomic real;
forall tlcu in tmpLCU do tlcu.write(0.0);
var tmpLCPC: [ClusNumSpace] atomic int;
forall tlcpc in tmpLCPC do tlcpc.write(0);

forall loc in Locales {
  on loc do {
    cobegin {
      forall (i,j) in ClusterSpace {
        tmpLCU[i,j].add(localCenterUpdates[i,j].read());
      }
      forall i in ClusNumSpace {
        tmpLCPC[i].add(localCenterPCounts[i].read());
      }
    }
  }
}
```

Note the use of nested forall and cobegin constrcuts. Also, the tmpLCU and tmpLCPC arrays are not distributed, yet Chapel gives seamless access to their elements even when the referring code runs on remote locales.

Finally, we test for convergence. If it has converged or the number of iterations has exceeded the given limit we will stop and will print results.

```
var b: atomic bool;
b.write(true);
forall (i,j) in ClusterSpace {
  var center: real = tmpLCU[i,j].read()/tmpLCPC[i].read();
  if (abs(center - currCenters[i,j]) > threshold){
    b.write(false);
  }
```

```
  currCenters[i,j] = center;
}
converged = b.read();
```

6.2 X10: K-Means Clustering

The following code is taken from X10 distribution and reproduced here with addition of comments. It is presented in segments highlighting only the details relevant to the idioms for clarity. Full working sample is presented in Appendix B.

- Declaration of points, dimension of a point, iterations, and number of clusters

```
static val DIM=2;
static val CLUSTERS=4;
static val POINTS=2000;
static val ITERATIONS=50;
```

- Definition of two dimensional points region.

```
static val points_region = 0..(POINTS-1)*0..(DIM-1);
```

- Creation of place local handles. Given a set of places in, the standard X10 library provides functionality to create an abstract reference, i.e. a place local handle that resolves to a local object in the particular place when referred. The place at which it will be referred need to be in the original set of places over which the place local handle is defined. The usage of place local handle is to create a distributed data structure, which can keep state local to places.

```
// random generator local to each place
val rnd = PlaceLocalHandle.make[Random](PlaceGroup.WORLD, () => new Random(0));
// local copy of current center points
val local_curr_clusters = PlaceLocalHandle.make[Array[Float](1)](
                PlaceGroup.WORLD, () => new Array[Float](CLUSTERS*DIM));
// sum of local points nearest to each center
val local_new_clusters = PlaceLocalHandle.make[Array[Float](1)](
                PlaceGroup.WORLD, () =>  new Array[Float](CLUSTERS*DIM));
// local count of points assigned to each center
val local_cluster_counts = PlaceLocalHandle.make[Array[Int](1)](
                PlaceGroup.WORLD, ()=> new Array[Int](CLUSTERS));
```

  Note. `PlaceGroup.WORLD` refers to all the places defined for the X10 program. Also, `=>` notation indicates anonymous functions, which are used to instantiate local instances. The `Array[Float](1)` and `Array[Int](1)` indicate one dimensional arrays of floating point numbers and integers.

- Distributed array to represent data points.

```
// domain decomposition by blocking along the zero^th dimension, i.e. along 0..(POINTS-1)
val points_dist = Dist.makeBlock(points_region, 0);
// creates a distributed array to represent data points
val points = DistArray.make[Float](points_dist, (p:Point)=>rnd().nextFloat());
```

  First line creates a block distribution along the zeroth axis of points region. Second line creates the actual array and initializes each element to a random value. Note. the `rnd()` refers to the place local `Random` object.

- Start with first four points as initial cluster centers

```
// global cluster centers
val central_clusters = new Array[Float](CLUSTERS*DIM, (i:int) => {
   // take the first four points as initial centers
   val p = Point.make([i/DIM, i%DIM]);
   return at (points_dist(p)) points(p); });
```

- The next steps are to refine cluster centers iteratively. Each round of refinement starts by copying the cluster centers found so far to `local_curr_clusters` arrays at each place and resetting `local_new_clusters` and

`local_cluster_counts` in those places. The resetting of place local handles is done in parallel at each place they are defined.

```
/* reset state */
finish {
    // foreach place, d, where points are distributed do in a new task
    for (d in points_dist.places()) async at(d) {
        async {
            for (var j:Int=0 ; j<DIM*CLUSTERS ; ++j) {
                // copy the current centers to the local copy of current centers
                local_curr_clusters()(j) = central_clusters(j);
                // reset new centers to origin
               local_new_clusters()(j) = 0;
            }
        }
        async {
            for (var j:Int=0 ; j<CLUSTERS ; ++j) {
               // reset point count assigned to each center as zero
               local_cluster_counts()(j) = 0;
            }
        }
    }
}
```

Note the use of `finish` construct to wait till remote activities complete above.

Each point is then checked against each cluster center for the Euclidean distance and is assigned to the center with the minimum distance. The comparison of each point with centers is trivially parallel and is done using remote activities operating at the place of particular point. The outline of the code performing this is given below.

```
/* compute new clusters and counters */
finish {
   // foreach point
    for (var p_:Int=0 ; p_<POINTS ; ++p_) {
        // do in a new task at the place of pth point
        async at(points_dist(p,0)) {
            // foreach cluster center compute the Euclidean
            // distance (square) from this point and assign
            // the point to the center with the closest distance
            for (var k:Int=0 ; k<CLUSTERS ; ++k) {
                // sum of squared components (square of Euclidean distance)
                // closest check
            }
            // add point to the nearest center points group
            // increment the point count for the particular center
        }
    }
}
```

The final step is to combine the place local new cluster centers and check for convergence. The copying of local centers to a global structure in one place accompanies careful use of place shifting of X10 using `at` construct. The idea is to run a remote activity at each place, which place shifts to a designated place and make an atomic update of local centers to global centers array.

```
finish {
    // put global centers and center poin counts to X10 Global Ref
    val central_clusters_gr = GlobalRef(central_clusters);
    val central_cluster_counts_gr = GlobalRef(central_cluster_counts);
```

```
        // the place where this code is executing
        val there = here;
    // foreach place, d, where points are distributed do in a new task at d
    for (d in points_dist.places()) async at(d) {
        // access PlaceLocalHandles 'here' and then data will be captured
        // by 'at' and transfered to 'there' for accumulation
        val tmp_new_clusters = local_new_clusters();
        val tmp_cluster_counts = local_cluster_counts();
        // local points for each center and counts are transferred to 'there'
        // and the global ref arrays are updated
        at (there) atomic {
            for (var j:Int=0 ; j<DIM*CLUSTERS ; ++j) {
                central_clusters_gr()(j) += tmp_new_clusters(j);
            }
            for (var j:Int=0 ; j<CLUSTERS ; ++j) {
                central_cluster_counts_gr()(j) += tmp_cluster_counts(j);
            }
        }
    }
}
```

Note how place local clusters and counts are referred to by temporary variables `tmp_new_clusters` and `tmp_cluster_counts`. These arrays are then automatically copied to `there` when doing the `at(there)` place shifting. Atomic blocks are used in updating the global cluster centers and counts.

The remaining code is trivial and self-explanatory. Its task is to test for convergence and print the new cluster centers if converged or if the total number iterations has completed.

6.3 Fortress: K-Means Clustering

We explain the Fortress implementation of k-means clustering in segments below. The complete code sample is given in Appendix C.

- Declaration of points, dimension of a point, iterations, and number of clusters.

```
numPoints: ZZ32 = 20
numDims: ZZ32 = 3
numClusters: ZZ32 =  4
numIter:ZZ32 = 50
threshold: RR64 = 0.0001
```

- Arrays to hold points, current centers, updated centers, updated point counts for each center.

```
points: Array[\RR64, (ZZ32,ZZ32)\] = array[\RR64\]((numPoints,numDims)).fill(rand)
currCenters: Array[\RR64, (ZZ32,ZZ32)\]
  = array[\RR64\]((numClusters,numDims)).fill(0.0)
newCenters: Array[\RR64, (ZZ32,ZZ32)\] = array[\RR64\]((numClusters,numDims))
newCentersPCounts: Array[\ZZ32, ZZ32\] = array[\ZZ32\](numClusters).fill(0)
```

  Note. `rand` is a custom function that takes two dimensional index of an array element and return a random value between `0` and `1`.

- Set initial centers to first `N` points where `N` is the number of clusters.

```
for (i,j) <- currCenters.indices() do
  newCenters[i,j] := points[i,j]
end
```

  Note. `for` loops in Fortress are parallel by default.

- Next steps happen iteratively in refining the centers. We start with resetting current centers and center updates as below.

```
do
   for (i,j) <- currCenters.indices() do
     currCenters[i,j] := newCenters[i,j]
     newCenters[i,j] := 0.0
   end
 also do
   for i <- 0#numClusters do
     newCentersPCounts[i] := 0
   end
 end
```

Note the use of `also` construct to run the two `for` loops in parallel.

Then we compare each point against current centers and assign them to the nearest centers.

```
for p <- 0#numPoints do
  closestC: ZZ32 = -1
  closestDist: RR64 = MaxD
  for c <- 0#numClusters do
    dist: RR64 = 0.0
    for d <- sequential(0#numDims) do
      dist +=  (points[p,d] - currCenters[c,d])^2
    end
    atomic do (* why atomic? because check for each cluster happens in parallel *)
      if (dist < closestDist) then
        closestDist := dist
        closestC := c
      end
    end
  end
  atomic do
    for d <- 0#numDims do
      newCenters[closestC,d] += points[p,d]
    end
    newCentersPCounts[closestC] += 1
  end
end
```

Finally, we check for convergence and update centers.

```
converged := true
for (i,j) <- newCenters.indices() do
   newCenters[i,j] /= newCentersPCounts[i]
   if |newCenters[i,j] - currCenters[i,j]| > threshold then
     atomic do
       converged := false
     end
   end
end
```

7. Summary

Parallel solutions are becoming the norm with increasing data in scientific analyses, yet HPC systems may not provide a ready for use solution. Higher level languages and programming libraries provide abstractions to assist programmers to compose parallel solutions without having to deal with intricacies of underlying the system. Memory model, in particular, is a reasonable classifier of current parallel programming support. The three distinct categories resulting from such classification are shared memory, distributed memory, and partitioned global address space.

Shared memory implementations present a unified layer of address space for the parallel tasks whereby allowing them to share data without interaction. This model is generally suited for intra-node parallelism where physically shared memory is available. Two of the widely known implementations of this type are OpenMP and Pthreads, where the former follows a fork-join model of threads while the latter follows a hierarchy of threads.

Distributed memory gives each task its own local address space and uses explicit communication among tasks to share data. This model fits well with the horizontally scalable architecture of HPC systems despite the imposed programmer obligations such as partitioning of data structures and explicit message passing. MPI falls into this category and is the *de facto* standard in communication among processes, which is generally used in a SPMD model of execution. An extension to the distributed memory model is to hybridize it with shared memory model which has become advantageous in the presence of multi-core processors.

Partition global address space model provides each process a local address space resembling distributed memory, yet shares a partitioned address space similar to shared memory among them. The shared memory layer avoids explicit message passing to share data while providing data locality through partitions. Similar to MPI, implementations of PGAS model follow an SPMD execution model where parallelism is fixed since startup. Direct realizations of PGAS model are the languages Unified Parallel C (UPC), Co-Array Fortran (CAF), and Titanium. HPCS languages as well adopts PGAS model for their memory abstraction, but do not expose it directly through the language.

An alternative classification of parallel programming support is the view of computing, which results two categories, i.e. local view and global view. Local view solutions comprise of separate cooperating tasks, which may operate on partitioned data structures. MPI and PGAS languages fall into this category though the latter has support for distributed data structures. Global view solutions, in contrast, start off with one task and adopt dynamic task creation to execute parallel code, while operating on global data structures. This model allows algorithms to adopt parallelism with minimal "shape" changes than with local view as mentioned in section 3. Implementations of this category include OpenMP and HPCS languages.

Despite the pros and cons in each category, the languages and libraries of parallel computing all intend to provide means to efficiently utilize HPC systems for scientific analyses. However, as hardware systems evolve other factors such as portability, programmability, and robustness become equally important. DARPA identifies these and performance as "productive" features of programming and intend the three HPCS languages, i.e. Chapel, X10, and Fortress, to be geared towards productivity rather than just performance. The common features of HPCS languages are dynamic task creation, global address space for data sharing, controlling data locality and mechanisms to synchronize tasks.

In addition to being productive, HPCS languages also need to be expressive enough to devise parallel solutions to large scale problems like scientific data analyses. It is common to notice patterns of code in parallel programs for tasks such as applying an operation in parallel on an array or spawning tasks to execute independent blocks of code. Expressive languages have built-in idioms for frequent patterns and this paper surveys five such idioms with respect to HPCS languages.

The first pattern is data parallel computations. Chapel provides the `forall` idiom, which is usable with single or multi-dimensional arrays, multiple arrays, or iterators without data in both statement and expression contexts. The runtime decides the degree of concurrency for the loop. X10's equivalent form composes of multiple constructs, i.e. `finish … for … async`, which may also be used to iterate over arrays or number sequences, but only within statement context. The `for` loop construct may be either `while` or `do-while` as well. Fortress loops are parallel by default; hence the simple `for` construct is a sufficient idiom, which supports iterations with or without data similar to Chapel's and X10's idioms. Also, it is an expression with void return type as Fortress is an expression oriented language.

The second set of idioms is for data distribution. Chapel introduces the unit locale to abstract the target environment that an execution would take place. Also it separates the index set, i.e. domain, from data elements of an array. Distribution of data elements over locales is in fact done by distributing the domain over locales and having data buckets in each place. The idiom to distribute a domain is `dmapped` followed by the specification of distribution type such as block, cyclic, and block-cyclic. X10 has a similar approach where the index set is named as the region and is spread out using the distribution of choice. The `DistArray` idiom is then used to define distributed arrays over the region. Fortress's approach differs slightly where the default behavior is to distribute data structures such as arrays, vectors, and matrices.

Asynchronous remote task creation is the third pattern. Chapel's solution is to use the constructs `on … begin` where the programmer specifies the locale and code to be run there asynchronously. X10 provides similar constructs, `at … async`, where the place of execution is given after the `at` construct. Fortress's abstraction of target environment slightly differs from Chapel and X10 where a region Fortress is bound to a particular element of the system's hierarchy such as a node, CPU, or core. However, it still provides a similar idiom, i.e. `spawn … at`. Also, notice that moving computation to a remote location and creating an asynchronous task are done through two constructs in all three languages. This make, it possible to interchange the order of operations to suit the particular need. As an example, in X10, `at (p) async S` and `async at(p) S` are both legal, but have different semantics.

The fourth kind is nested parallelism, which means combining data parallel operations with task parallel operations. Chapel's idiom is to use `cobegin … forall`, where `cobegin` is the task parallel and `forall` is data parallel. These can be nested arbitrary as necessary. X10's approach is to use `for … async` constructs, where data parallel code may be constructed using the `for` construct and task parallelism may be done with `async`. Similar to X10, Fortress provides the `for … spawn` constructs.

The final pattern is performing remote transactions. Chapel is still in development stages with respect to this, but intends to provide an idiom of the form `on … atomic`. X10 has a similar yet working implementation of the form `at … atomic`. X10's atomicity is intentionally made to be not absolute for performance reasons. Fortress too has a working version, `at … atomic` as well.

The sample k-means implementations demonstrate the use of above idioms in practice. All three implementations look alike in their structure, except data distribution is not present in Fortress's implementation due to lack of runtime support. The Chapel and X10 versions also give insight to some of the distributions available in the languages.

In conclusion, HPC systems evolve and HPCS languages intend to provide a language level solution to the productive development of parallel programs. They bring out the best of data locality and data sharing together with global view of computing. At the same time they provide a rich set of idioms to frequent patterns in parallel programming, which makes them preferable over existing technologies for the development of data and compute intensive applications in years to come.

Acknowledgements

The author would like to thank Geoffrey Fox for many discussions and advices on the survey and Amr Sabry for his insight on language comparison. Also thanks to Kent Dybvig and Haixu Tang for their support as advisory members.

8. References

[1] Fortress Language Specification.

[2] BINGJING, Z., YANG, R., TAK-LON, W., QIU, J., HUGHES, A., and FOX, G., 2010. Applying Twister to Scientific Applications. In Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, 25-32. DOI= http://dx.doi.org/10.1109/CloudCom.2010.37.

[3] CHAMBERLAIN, B.L., CALLAHAN, D., and ZIMA, H.P., 2007. Parallel Programmability and the Chapel Language. Int. J. High Perform. Comput. Appl. 21, 3, 291-312. DOI= http://dx.doi.org/10.1177/1094342007078442.

[4] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., PRAUN, C.V., and SARKAR, V., 2005. X10: an object-oriented approach to non-uniform cluster computing. In Proceedings of the Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (San Diego, CA, USA2005), ACM, 1094852, 519-538. DOI= http://dx.doi.org/10.1145/1094811.1094852.

[5] CULLER, D.E., SINGH, J.P., and GUPTA, A., 1999. Parallel computer architecture : a hardware/software approach. Morgan Kaufmann Publishers, San Francisco.

[6] DEAN, J. and GHEMAWAT, S., 2004. MapReduce: Simplified Data Processing on Large Clusters. Sixth Symposium on Operating Systems Design and Implementation, 137-150.

[7] GROPP, W., 2001. Learning from the Success of MPI. In Proceedings of the Proceedings of the 8th International Conference on High Performance Computing (2001), Springer-Verlag, 652928, 81-94.

[8] GUY STEELE, J.-W.M., 2006. Fortress Programming Language Tutorial. In ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Canada.

[9] INC, C., Chapel Language Specification, Version 0.92

[10] JACK DONGARRA, IAN FOSTER, GEOFFREY FOX, WILLIAM GROPP, KEN KENNEDY, LINDA TORCZON, and ANDY WHITE, 2002. The Sourcebook of Parallel Computing. Morgan Kaufmann.

[11] LUSK, E. and YELICK, K., 2007. LANGUAGES FOR HIGH-PRODUCTIVITY COMPUTING: THE DARPA HPCS LANGUAGE PROJECT. Parallel Processing Letters 17, 01, 89-102. DOI= http://dx.doi.org/doi:10.1142/S0129626407002892.

[12] PROKOWICH, S.J.D.B.L.C.S.-E.C.D.I.L., 2010. Five Powerful Chapel Idioms. In CUG 2010, Edinburgh.

[13] VIJAY SARASWAT, B.B., IGOR PESHANSKY, OLIVIER TARDIEU, AND DAVID GROVE, 2012. X10 Language Specification.

[14] WEILAND, M., 2007. Chapel, Fortress and X10: Novel Languages for HPC.

[15] YANG RUAN, C.A., Parallelism for LDA.

9. Appendix A

Complete working code of k-means clustering written in Chapel.

```
use BlockDist,ReplicatedDist;
use Random;
config var numDim: int = 2,
           numClusters: int = 4,
           numPoints: int  = 2000,
           numIter: int = 50,
           threshold: real = 0.0001;

proc main() {
  writeln ("dimensions: ", numDim);
  writeln ("clusters ", numClusters);
  writeln ("points: ", numPoints);
  writeln ("iterations: ", numIter);
  writeln ("threshold: ", threshold);

  /* domains */
  const PointsSpace = {0..#numPoints,0..#numDim};
  const ClusterSpace = {0..#numClusters,0..#numDim};
  const ClusterNumSpace = {0..#numClusters};

  /* distribute points and initialize to random values */
  var blockedLocaleView = {0..#numLocales,1..1};
  var blockedLocales: [blockedLocaleView] locale = reshape(Locales, blockedLocaleView);
  // this in essence blocks along the zeroth dimension (i.e. range 0..#numPoints)
  const BlockedPointsSpace = PointsSpace dmapped Block(boundingBox=PointsSpace,
                                                       targetLocales=blockedLocales);
  var points: [BlockedPointsSpace] real;
  var rand: RandomStream = new RandomStream();
  rand.fillRandom(points);

  /* current centers initilized to first N points where N = numClusters */
  var currCenters: [ClusterSpace] real;
  currCenters = points[ClusterSpace];

  writeln("initial centers");
  writeln(currCenters);

  /* replicated local center */
  const ReplClusterSpace = ClusterSpace dmapped ReplicatedDist();
  var localCurrCenters: [ReplClusterSpace] real;
  // work around to non existing atomic blocks
  var localCenterUpdates: [ReplClusterSpace] atomic real;
  const ReplClusterNumSpace = ClusterNumSpace dmapped ReplicatedDist();
  var localCenterPCounts: [ReplClusterNumSpace] atomic int;

  var converged: bool = false;
  var step: int = 0;
  // whish something like real.max exist
  const MaxDist = 9223372036854775807;
  while (step < numIter && !converged) {
    cobegin {
      // copy currCenters to each replicand
      localCurrCenters = currCenters;
      // reset local updates and counts
      forall lcu in localCenterUpdates do lcu.write(0.0);
      forall lcpc in localCenterPCounts do lcpc.write(0);
```

```
}

forall p in {0..#numPoints} {
  on points[p,0] {
    var closestC: int = -1;
    var closestDist: real = MaxDist;
    for c in {0..#numClusters} { // possibility to parallelize
      var dist: atomic real;
      dist.write(0.0);
      forall d in {0..#numDim} {
        var tmp = points[p,d] - localCurrCenters[c,d];
        dist.add(tmp*tmp);
      }

      if (dist.read() < closestDist) {
        closestDist = dist.read();
        closestC = c;
      }
    }

    forall d in {0..#numDim} {
      atomic { // would have been great, but not implemented yet in Chapel
        localCenterUpdates[closestC,d].add(points[p,d]);
      }
    }
    localCenterPCounts[closestC].add(1);
  }
}

var tmpLCU: [ClusterSpace] atomic real;
forall tlcu in tmpLCU do tlcu.write(0.0);
var tmpLCPC: [ClusNumSpace] atomic int;
forall tlcpc in tmpLCPC do tlcpc.write(0);

forall loc in Locales {
  on loc do {
    cobegin {
      forall (i,j) in ClusterSpace {
        tmpLCU[i,j].add(localCenterUpdates[i,j].read());
      }
      forall i in ClusNumSpace {
        tmpLCPC[i].add(localCenterPCounts[i].read());
      }
    }
  }
}

var b: atomic bool;
b.write(true);
forall (i,j) in ClusterSpace {
  var center: real = tmpLCU[i,j].read()/tmpLCPC[i].read();
  if (abs(center - currCenters[i,j]) > threshold){
    b.write(false);
  }
  currCenters[i,j] = center;
}
```

```
    converged = b.read();
    step += 1;
  }

  writeln("final centers");
  writeln(newCenters);


}
```

10. Appendix B

Complete code sample for k-means clustering from the X10 distribution with few modifications.

```
import x10.io.Console;
import x10.util.Random;

public class KMeansDist {

    static val DIM=2;
    static val CLUSTERS=4;
    static val POINTS=2000;
    static val ITERATIONS=50;

    static val points_region = 0..(POINTS-1)*0..(DIM-1);

    public static def main (Array[String]) {
        // random generator local to each place
        val rnd = PlaceLocalHandle.make[Random](PlaceGroup.WORLD, () => new Random(0));

        // local copy of current center points
        val local_curr_clusters = PlaceLocalHandle.make[Array[Float](1)](
                    PlaceGroup.WORLD, () => new Array[Float](CLUSTERS*DIM));

        // sum of local points nearest to each center
        val local_new_clusters = PlaceLocalHandle.make[Array[Float](1)](
                    PlaceGroup.WORLD, () =>  new Array[Float](CLUSTERS*DIM));

        // local count of points assigned to each center
        val local_cluster_counts = PlaceLocalHandle.make[Array[Int](1)](
                    PlaceGroup.WORLD, ()=> new Array[Int](CLUSTERS));

        // domain decomposition by blocking along the zeroth dimension,
        // i.e. along 0..(POINTS-1)
        val points_dist = Dist.makeBlock(points_region, 0);

        // creates a distributed array to represent data points
        val points = DistArray.make[Float](points_dist, (p:Point)=>rnd().nextFloat());

        // global cluster centers
        val central_clusters = new Array[Float](CLUSTERS*DIM, (i:int) => {
             // take the first four points as initial centers
             val p = Point.make([i/DIM, i%DIM]);
            return at (points_dist(p)) points(p);
        });

        // global count of points assigned to each cluster center
        val central_cluster_counts = new Array[Int](CLUSTERS);
```

```
// keeps the previous set of centers during iterations (global)
val old_central_clusters = new Array[Float](CLUSTERS*DIM);



/* refine cluster centers */
for (i in 1..ITERATIONS) {

    Console.OUT.println("Iteration: "+i);

    /* reset state */
    finish {
        // foreach place, d, where points are distributed do in a new task
        for (d in points_dist.places()) async at(d) {
            async {
                for (var j:Int=0 ; j<DIM*CLUSTERS ; ++j) {
                    // copy the current centers to the local copy
                    local_curr_clusters()(j) = central_clusters(j);
                    // reset new centers to origin
                    local_new_clusters()(j) = 0;
                }
            }
            async {
                for (var j:Int=0 ; j<CLUSTERS ; ++j) {
                    // reset point count assigned to each center as zero
                    local_cluster_counts()(j) = 0;
                }
            }
        }
    }

    /* compute new clusters and counters */
    finish {
     // foreach point
        for (var p_:Int=0 ; p_<POINTS ; ++p_) {
            val p = p_;
            // do in a new task at the place of pth point
            async at(points_dist(p,0)) {
                var closest:Int = -1;
                var closest_dist:Float = Float.MAX_VALUE;
                // foreach cluster center compute the Euclidean
                // distance (square) from this point and assign
                // the point to the center with the closest distance
                for (var k:Int=0 ; k<CLUSTERS ; ++k) {
                    var dist : Float = 0;
                    // sum of squared components (square of Euclidean distance)
                    for (var d:Int=0 ; d<DIM ; ++d) {
                        val tmp = points(Point.make(p,d))
                                    - local_curr_clusters()(k*DIM+d);
                        dist += tmp * tmp;
                    }
                    // closest check
                    if (dist < closest_dist) {
                        closest_dist = dist;
                        closest = k;
```

```
                }
            }
            // it's possible for two activities in same place to
            // compete for the increment operations below.
            // So need to use atomic
            atomic {
                // add point to the nearest center points group
                for (var d:Int=0 ; d<DIM ; ++d) {
                    local_new_clusters()(closest*DIM+d) +=
                                            points(Point.make(p,d));
                }
                // increment the point count for the particular center
                local_cluster_counts()(closest)++;
            }
        }
    }
}


// copy gloabl centers to old global centers and clear global centers
for (var j:Int=0 ; j<DIM*CLUSTERS ; ++j) {
    old_central_clusters(j) = central_clusters(j);
    central_clusters(j) = 0;
}

// clear global center point counts
for (var j:Int=0 ; j<CLUSTERS ; ++j) {
    central_cluster_counts(j) = 0;
}

finish {
 // put global centers and center poin counts to X10 Global Ref
    val central_clusters_gr = GlobalRef(central_clusters);
    val central_cluster_counts_gr = GlobalRef(central_cluster_counts);

    // the place where this code is executing
    val there = here;

    // foreach place, d, where points are distributed do in a new task at d
    for (d in points_dist.places()) async at(d) {
        // access PlaceLocalHandles 'here' and then data will be captured
        // by 'at' and transfered to 'there' for accumulation
        val tmp_new_clusters = local_new_clusters();
        val tmp_cluster_counts = local_cluster_counts();

        // local points for each center and counts are transferred to 'there'
        // and the global ref arrays are updated
        at (there) atomic {
            for (var j:Int=0 ; j<DIM*CLUSTERS ; ++j) {
                central_clusters_gr()(j) += tmp_new_clusters(j);
            }
            for (var j:Int=0 ; j<CLUSTERS ; ++j) {
                central_cluster_counts_gr()(j) += tmp_cluster_counts(j);
            }
        }
    }
```

```
            }

            // by now  central_clusters should have received all the sums
            // of poins for each cluster
            for (var k:Int=0 ; k<CLUSTERS ; ++k) {
                for (var d:Int=0 ; d<DIM ; ++d) {
                    // take the mean of points for each cluster as the global centers
                    central_clusters(k*DIM+d) /= central_cluster_counts(k);
                }
            }

            // test for convergence
            var b:Boolean = true;
            for (var j:Int=0 ; j<CLUSTERS*DIM ; ++j) {
                if (Math.abs(old_central_clusters(j)-central_clusters(j))>0.0001) {
                    b = false;
                    break;
                }
            }
            if (b) break;

        }

        // print the new centers after convergence
        for (var d:Int=0 ; d<DIM ; ++d) {
            for (var k:Int=0 ; k<CLUSTERS ; ++k) {
                if (k>0)
                    Console.OUT.print(" ");
                Console.OUT.print(central_clusters(k*DIM+d));
            }
            Console.OUT.println();
        }
    }
}
```

11. Appendix C

Complete code sample of k-means clustering implemented in Fortress.

```
component KMeans
export Executable

rand(i:ZZ32,j:ZZ32):RR64=random(1.0)

run():()=do
  numPoints: ZZ32 = 20
  numDims: ZZ32 = 3
  numClusters: ZZ32 =  4
  numIter:ZZ32 = 50
  threshold: RR64 = 0.0001

  points: Array[\RR64, (ZZ32,ZZ32)\] = array[\RR64\]((numPoints,numDims)).fill(rand)
  currCenters: Array[\RR64, (ZZ32,ZZ32)\] = array[\RR64\]((numClusters,numDims)).fill(0.0)
```

```
newCenters: Array[\RR64, (ZZ32,ZZ32)\] = array[\RR64\]((numClusters,numDims))
newCentersPCounts: Array[\ZZ32, ZZ32\] = array[\ZZ32\](numClusters).fill(0)

for (i,j) <- currCenters.indices() do
  newCenters[i,j] := points[i,j]
end

println("initial centers")
println(newCenters)
println()

MaxD: RR64 = 9223372036854775807
converged: Boolean = false
step: ZZ32 = 0

while ((NOT converged) AND (step < numIter)) do
  do
    for (i,j) <- currCenters.indices() do
      currCenters[i,j] := newCenters[i,j]
      newCenters[i,j] := 0.0
    end
  also do
    for i <- 0#numClusters do
      newCentersPCounts[i] := 0
    end
  end

  for p <- 0#numPoints do
    closestC: ZZ32 = -1
    closestDist: RR64 = MaxD
    for c <- 0#numClusters do
      dist: RR64 = 0.0
      for d <- sequential(0#numDims) do
        dist +=  (points[p,d] - currCenters[c,d])^2
      end

      atomic do (* why atomic? because check for each cluster happens in parallel *)
        if (dist < closestDist) then
          closestDist := dist
          closestC := c
        end
      end
```

```
      end

      atomic do
        for d <- 0#numDims do
          newCenters[closestC,d] += points[p,d]
        end
        newCentersPCounts[closestC] += 1
      end
    end

    converged := true
    for (i,j) <- newCenters.indices() do
      newCenters[i,j] /= newCentersPCounts[i]
      if |newCenters[i,j] - currCenters[i,j]| > threshold then
        atomic do
          converged := false
        end
      end
    end

    step += 1
  end

  println("final centers")
  println(newCenters)
end (*run*)
end (*component*)
```