



# Chapel, Fortress and X10: novel languages for HPC

Michèle Weiland

*EPCC, The University of Edinburgh, James Clerk Maxwell Building,  
Mayfield Road, Edinburgh, EH9 3JZ, UK*

October 10, 2007

## **Abstract**

Chapel, Fortress and X10 are novel languages focussed on the HPC community. They have been developed with the aim to facilitate the programming of large next-generation parallel systems and increase both the productivity of the programs' developers and the scalability of the developed codes. This report introduces these languages by offering an overview of the design and specification of each language. The report will focus on each language's way of handling task and data parallelism. At the time of writing, all three languages are still in early development stages and any available compilers are experimental. For this reason (in part also for licensing reasons), the report does not touch on language or code performance.

**This is a Technical Report from the HPCx Consortium**

**© HPCx UoE Ltd 2007**

Neither HPCx UoE Ltd nor its members separately accept any responsibility for loss or damage from the use of information contained in any of their reports or in any communication about their tests or investigations.

## 1 Introduction

In 2002, DARPA<sup>1</sup> launched their High Productivity Computing Systems (HPCS) programme, offering funding for industry and academia to research the development of computing systems which focus on high productivity along with high performance. Part of this programme concentrates on the specification of novel languages for the HPC community. In Phase 2 of the programme (July 2003 - July 2006) the three remaining partners that were awarded funding were Cray, IBM and SUN<sup>2</sup>; SUN were eventually dropped from the programme at the start of Phase 3.[1]

This report will introduce the novel programming languages *Chapel* (Cray), *Fortress* (SUN) and *X10* (IBM) that were developed as part of the HPCS programme.

## 2 Aims and Motivation of HPCS

A major challenge for modern HPC systems is their lack of programmability. Even though the size and speed of modern HPC systems keeps increasing, it becomes ever more difficult to exploit all the resources such a system has to offer. The efficiency with which the systems are programmed and used decreases as a result, and their productivity often remains below an optimal level. The goal of the HPCS programme is to develop a systems that improves the overall productivity of high performance computing. This also includes widening the pool of developers that have the knowledge needed to program parallel machines: currently only a small subset of those programmers who have the ability to develop high quality serial code.

The HPCS programme defines productivity as “a combination of performance, programmability, portability and robustness”. The languages developed under HPCS should try to address the improvement of the programmability and overall productivity of next generation parallel machines. The languages should support general parallelism and separate algorithms from implementation, i.e. data and task distribution should be independent of the tasks and the computations on the data.

## 3 Overview of the Basic Language Concepts

The following paragraphs will give a general overview of the three languages. Subsequent sections will look in more detail at each language’s specification of parallelism, especially the handling of task parallelism and distributed data structures, and code examples for each language.

### 3.1 Fortress<sup>3</sup>

Fortress is SUN’s effort in the HPCS programme; it was developed with the intent to produce a “secure Fortran” (from which it derives its name). Even though many of

---

<sup>1</sup>Defense Advanced Research Projects Agency, a U.S Department of Defense institute.

<sup>2</sup>SGI and HP were part of Phase 1, but did not receive the funding to continue their research.

<sup>3</sup>See [2]

the aspects of Fortress are based on existing programming languages, its syntax is new and designed to be as close to mathematical notation as possible. SUN believes that, because mathematical notation is understood by scientists regardless of their computing background, using this type of syntax will make programs more robust and readable. The actual code that represents a certain equation should thus closely resemble the mathematical notation of this equation. A simple example of how Fortress emulates mathematical notation is the use of juxtaposition to represent multiplication. The multiplication operator  $*$  can be defined, but is not necessary. The expression  $a = b c$  is therefore equivalent to  $a = b * c$ . Fortress takes the approach further by introducing the static checking of physical units and allowing the use of unicode characters. The Fortress library defines a set of default dimensions and units, which (together with the unicode characters) can be used in the definitions of variables:

```

l : ℝ Length = 5.5 m
v : ℝ3 Velocity = [1.1 0 5.6] m/s

```

In ASCII, the two definitions are written as follows:

```

l:RR Length = 5.5 m_
v:RR^3 Velocity = [1.1 0 5.6] m_/s_

```

Fortress' syntax allows users easily to express functions such as sums or products over sets of elements. For instance, a function that defines a sum over  $n$  elements,  $f(n) = \sum_{i=1}^n i$ , is represented as `f(n) = SUM[i <- 1:n] i`. Fortress also supports the definition of *comprehensions*; they describe collections of elements using rules that are valid for all elements. Reduction operations are also implemented as part of the language standard.

The basic building blocks of Fortress code are *objects* and *traits*. Objects define *fields* and *methods*, whereas traits declare sets of methods<sup>4</sup>. Traits can declare abstract (i.e. header only) or concrete (i.e. header and definitions) methods. Fortress is an interpreted language; the interpreter currently runs on the JVM and implements only a small fraction of the language specification.

### 3.2 X10<sup>5</sup>

The name X10 is derived from the developers' aim to create, by 2010, a language that is ten 10 times more productive than those languages and libraries that are commonly used for current HPC codes. IBM's HPCS effort X10 uses a very different approach to the design of a high productivity language for parallel systems. Instead of trying to design an entirely new language with new syntax, types, data structures and parallelism constructs, X10 is defined as an extension of Java and the serial subsets of both languages are very similar. The design goals for X10 are to create a language which enables the development of safe, scalable, analysable and flexible code. Java was chosen as a basis for X10 because it supports three out of these four goals. It is also a widespread and accepted modern OO language that is accompanied by reliable tools, libraries and

---

<sup>4</sup>Traits in Fortress are similar to interfaces in Java

<sup>5</sup>See [3]

documentation. X10 can be seen as Java minus concurrency, arrays or built-in types, plus *places*, *activities*, *clocks*, *distributes multi-dimensional arrays* and *value types*. These X10 specific, parallelism supporting constructs will be introduced in Sections 4 and 5.

At the time of writing only a preliminary version of the X10 compiler (which compiles the X10 language to Java) is available and only a subset of the language specification is implemented<sup>6</sup>. The distribution includes a development toolkit, X10DT, which runs on Eclipse 3.1.2 and Java 5 VM.

### 3.3 Chapel<sup>7</sup>

The name Chapel stands for Cascade<sup>8</sup> High Productivity Language. Similarly to Fortress, Cascade is not directly built on an existing language like C or Fortran; the developers believe that too much similarity between Chapel and an existing sequential language could lead to confusion and would encourage a “sequential” programming style. Chapel’s syntax borrows from a set of languages, namely C, Fortran, Java and Ada. A Chapel program is divided into *modules* and use two types of classes to support OO features: *traditional classes*, which are passed by reference (similar to classes in Java), and *value classes*, which are passed by value (similar to structures or classes in C++).

Chapel defines three types of variables: *var*, *const*, which defines runtime constants, and *param*, defining compile-time constants. Chapel also introduces a number of new types, including *ranges*, *domains*, *enumerations* and *tuples*. An enumeration defines a set of named, unordered constants; a tuple is an ordered set of elements, which can be of any type. Ranges simply define arithmetic sequences, for example:

```
var r : range = 1..n;
```

A domain on the other hand defines a set of indices. Three types of domains are possible, namely arithmetic, indefinite and opaque domains. An arithmetic domain denotes indices in multiple dimension, for example *domain(3)* is used to define a three dimensional index set:

```
var d : domain(3) = [1..i, 1..j, 1..k];
```

In opaque domains, the indices do not need to have relations to one another, and each index is unique. These domains can be used to represent graphs, sets and linked lists. An indefinite domain can represent indices of user-defined, arbitrary types and can be used to implement hash table functionality or associative arrays. An example of the use of an indefinite domain is [5]:

```
var People: domain(string);  
var Age: [People] int;  
People += "John";
```

---

<sup>6</sup>The X10 distribution is available from <http://x10.sourceforge.net/x10downloads.shtml>.

<sup>7</sup>See [4]

<sup>8</sup>Cascade is the system Cray is developing under HPCS.

```
Age("John") = 22;
```

Domains are an important feature of Chapel and are the basis to array definition, manipulation and distribution. Details of this will be presented later in this report.

A preliminary Chapel compiler, which translates the Chapel code into C, is currently not available for public download. In order to receive a copy of the compiler, users need to contact Cray with a statement of interest to be considered for future limited releases.

## 4 Task Parallelism

The three languages are developed using different design principles with regard to their basic syntax and properties. However, with regard to their implementation of parallelism, they share a common approach of using global view programming with multi-threading to fit the architectures of next-generation parallel systems, which are likely to make use of multi-core SMP nodes in non-uniform clusters. The hierarchical structures of parallelism in the systems thus need to be easily representable in the code that is written for such systems. Additionally, it is widely believed that a fragmented memory model, which relies on making processors communicate via message-passing, reduces developer productivity. Instead, the languages are based on a global view model, using a single partitioned address space and thus permitting the code developers to think of a single computation running across multiple processors. The following paragraphs will explain how Fortress, X10 and Chapel address this problem and try to offer flexible and robust solutions.

### 4.1 Fortress

Fortress was developed as a multi-purpose language, with an emphasis on large parallel systems. One of the design goals was to facilitate the development of code for these systems and the core of the language provides a built-in support for both task and data parallelism. Task parallelism in Fortress is managed using two types of *threads*: *implicit* threads and *spawned* (or explicit) threads. Built-in parallel constructs use implicit threads to achieve parallelism. The implicitly parallel constructs include: parallel *for* loops; the *do ... also do ... end* construct; and tuple expressions. In Fortress, *for* loops are parallel by default and thus the order of execution of the loop iterations is arbitrary. A tuple expression contains a set of elements or functions; these subexpression are evaluated in parallel. A similar parallel execution can be achieved by using the *do ... also do ... end* construct. For example, the function tuple (*fred(a)*, *fred(b)*, *fred(c)*) which describes the parallel computation of three *fred* functions, can also be expressed thus:

```
do
  fred(a)
also do
  fred(b)
also do
  fred(c)
end
```

A running Fortress program consists of a set of threads and a memory; the memory consists of a set of *locations*. These locations are placed in specified *regions*, which describe the underlying structure of the machine. The regions are organised hierarchically in a tree; the low levels of the tree describe the local entities, such as memory shared by a number of CPUs, and the higher levels of the tree structure represent entities that are distributed.

## 4.2 X10

Where Fortress uses regions and locations, X10 uses the notion of *places*, which represent computational units with local shared memory. Every X10 program runs over a set of places and each place can host data or run *activities*. Places are instances of the built-in class `x10.place.lang`; the number of places available can be specified from the command line. The initial X10 program starts in the first place (which can be accessed by reading `place.FIRST_PLACE`), a neighbouring place can for instance be returned using the operation `next()`. An X10 activity is a lightweight thread that can either run on the place to which it belongs, or (explicitly or implicitly) asynchronously update memory in other places. The statement

```
async (P) S
```

is executed at the place  $P$  by spawning an activity that executes statement  $S$ . A spawned thread can in return spawn other threads in different places. X10 also uses *atomic* statements to ensure access of locally available data only. In order to coordinate multiple process, X10 needs to use multiple barriers. This is achieved through the use of *clocks*: every activity has a number of clocks associated with them, which, at any given point of an execution, are in a given phase and thus keep track of the progress of the activity. The clock can only move to the next phase once the associated activities have completed their computations.

## 4.3 Chapel

*Locale* is what a computational unit for uniform memory access is called in Chapel. In a cluster architecture, this would be a shared memory node. Every running Chapel program is automatically provided with an array of locales (of the built-in locale type) that reflect the part of the machine on which the program is executed. These locales can be repartitioned and remapped by the code developers depending on the needs of the algorithms and computations.

Task parallelism is mainly supported through parallel statements such as *forall*, *begin* and *cobegin*, and the synchronization variables *single* (can only be assigned a value once in its lifetime) and *synch* (can be assigned values multiple times). With the *forall* statement the compiler is instructed that the operations or statements inside that loop are independent of each other and can be executed concurrently, for example:

```
forall i in 1..N do
    writeln(" i = ", i)
```

Computations can be spawned using either the *begin* or the *cobegin* statement. Their use is best illustrated using an example:

```
var finished = synch bool;
begin
    while(!finished) do stuff();
morestuff();
finished=true;

cobegin{
    fred(a);
    fred(b);
    fred(c);
}
```

The *begin* statement spawns a computation that, in an unstructured way, can be executed in parallel with the statements that follow it and control continues with the following statements. A statement spawned by *begin* is typically only executed for its side-effects. The *cobegin* statement on the other hand creates a block of statements that can be executed concurrently. Control only continues once all the computations inside the block are finished.

## 5 Data Parallelism and Array Manipulation

In order to achieve the goal of making the implementation of an algorithm independent of the data this algorithm manipulates, the languages need to have built-in support for parallel and distributed data structures. Rather than dividing data up into small parts and actively distributing them between the available processors, the aim is to maintain a global view of the data.

### 5.1 Fortress

The Fortress developers decided that a set of default data constructs should be supported by and built into the language. Therefore a novel type system was designed to provide a number of frequently used types, which are defined in standard libraries, similar to a language like Java. Among the default types are lists, vectors, sets, maps, matrices and multi-dimensional arrays. Most of the data structures, for instance arrays and matrices, contain *distributions*. The distributions divide large data structures and thus enable data parallelism. Built-in distributions include *sequential* (data structures are assigned one contiguous piece of memory), *par* (blocks of size one), or *blocked* (blocks of equal sizes). In Fortress two-dimensional arrays can be declared and used as follows:

```

a: RR64[10,10], b: RR64[10,10], sum: RR64[10,10]
for i<-0#10 do
  for j<-0#10 do
    sum[i,j] := a[i,j] + b[i,j]
  end
end
end

```

A matrix  $m$  can simply be declared and initialised thus:

```

m = [ 1 2 3
      4 5 6 ]

```

## 5.2 X10

Similarly to Chapel, X10 specifies two types of objects: reference objects and value objects. A reference object contains mutable fields and cannot be copied between different places of the machine. A value object on the other hand contains has no updatable fields and can be copied freely between places. In X10 primitive types are not built-in, but are provided in the standard library as value classes. Arrays are defined as the mapping of a *distribution* to a range type, or *region*, which specifies a set of indices. A two-dimensional region, i.e. a region with rank 2, with indices ranging 1 to  $i$  and  $j$  respectively, is defined thus:

```

region Reg = [1:i, 1:j];

```

This region can then be used in the declaration of a distribution (which can have sub-distributions), mapping the region to a set of places. Several types of distributions are built-in, including *block*, *cyclic* or *unique*. A distribution is then declared thus:

```

distribution Dist = distribution.factory.block(Reg);

```

Distributed arrays in X10 are functions from distributions, rather than types. For example:

```

int[Dist] array = new int [Dist] (point [i,j] ){return i * j;}

```

is the declaration of an integer array of the distribution `Dist`, and every point of the array is assigned the product of  $i$  and  $j$ . A number of operations can be performed on such arrays, including *reduction* operations and *scans*.

## 5.3 Chapel

Chapel uses arithmetic domains (see Section 3.3) in the definition of arrays:

```

var Dom: domain(2) = [1..m, 1..n];
var A: [Dom] int;
var B: [Dom] float;

```

Arrays  $A$  and  $B$  use the same index set, thus they can be defined using the same domain. *forall loops* are used to iterate in parallel over domains and arrays. It is possible to define *subdomains*, which represent a subset of the original domain. The following example shows a subdomain of the domain  $\text{Dom}$ , and an iteration over an array defined within that subdomain:

```

var smallDom: subdomain(Dom) = [2..m-1, 2..n-1];
var smallA: [smallDom] int;
forall ij in smallDom {
    smallA(ij) = value;
}

```

Chapel also introduces an *index* type, which can represent any set of indices inside a domain. The following two examples show how the index type can be used to store the indices of the centre or a slice of a domain:

```

var centre: index(Dom) = [m/2, n/2];
var slice: index(Dom) = [1,1..n];

```

Scalar operators can be promoted over domains and arrays and reduction operations and scans are supported. The following example shows the use of a max reduction over the absolute values in array  $A$  in order to collapse the result into a scalar:

```

var res = max reduce abs(A);

```

Domain, and thus arrays, can be distributed over multiple locales. Chapel supports distributions such as block and cyclic, but it also allows developers to define their own, machine-specific distributions. This is useful when data needs to be distributed in a way that is not reflected in the standard distributions. Domain distribution gives developers the possibility to control the locality of data.

## 6 Code Examples

The following examples are taken from the language distributions and are reproduced here (with slight modifications) to illustrate the languages. The first example in Figure 1 is a Fortress implementation of the Buffon's Needle algorithm: the algorithm is used to estimate the value of  $\pi$  by dividing the number of times the dropped needle hits a line by the total number of drops. The higher the number of 'drops' or iterations, the more accurate the estimate will be. The calculations in each iteration are independent of each other, therefore the loop can be executed in parallel. This is achieved through the implicitly parallel `for` loop (line14). The example illustrates a number of features of the Fortress language: line 9 (as well as lines 20 and 21) for instance shows how the

juxtaposition of elements is used to represent multiplication. It also shows the different declarations of variables that are possible, with mutable variables of specified types and immutable, constant, elements. The atomic expressions are used for concurrency control and make sure that only one thread at the time can read or write the variables within the expression.

```

1 component fortress.executable
2
3 export Executable
4
5 run(args:String...):()=do
6
7     needleLength = 20 // declaration of
8     numRows = 10 // immutable variables
9     tableHeight = needleLength numRows
10
11     var hits : RR64 = 0.0 // declaration of mutable
12     var n : RR64 = 0.0 // variables of type RR64
13
14     for i <- 1#3000 do // 3000 iterations
15         delta_X = random(2.0) - 1
16         delta_Y = random(2.0) - 1
17         rsq = delta_X^2 + delta_Y^2
18
19         if 0 < rsq < 1 then
20             y1 = tableHeight random(1.0)
21             y2 = y1 + needleLength (delta_Y / sqrt(rsq))
22             (y_L, y_H) = (y1 MIN y2, y1 MAX y2)
23
24             // increase 'hits' if needle hits line
25             if ceiling(y_L/needleLength) = floor(y_H/needleLength) then
26                 atomic do hits += 1.0 end
27             end
28             atomic do n += 1.0 end
29         end
30     end
31
32     probability = hits/n
33     pi_est = 2.0/probability
34     end
35 end

```

Figure 1: Fortress implementation of the Buffon's Needle algorithm, used for the estimation of  $\pi$ . The main iteration is a parallel loop, allowing the calculations to be executed concurrently.

Figure 2 shows an implementation of the Jacobi algorithm in Chapel. The grid size for the calculation is defined using two domains, and the problem spaces themselves are subsequently declared using these domains. The grid directions are represented using tuples of two values (line 17), which are added to any coordinates on the grid (line 22) to refer to the correct points around the current coordinate. Line 21 shows the parallel `forall` loop, which performs the main calculation for all coordinates  $ij$  inside the problem space concurrently. A reduction operation is then used to calculate the difference between the old and the new problem space. Figure 3 shows an implementation of the same algorithm in X10. Instead of using domains, the problem size is defined using regions and distributions (lines 7 to 14). The parallelisation of the calculation is achieved implicitly for every point  $[i, j]$  through the assignment to a distribution (lines 27 and 28).

```

1  config var n = 5,                // size of n x n grid
2      epsilon = 0.00001;         // convergence tolerance
3
4  def main() {
5      const ProblemSpace = [1..n, 1..n], // domain for grid points
6          BigDomain = [0..n+1, 0..n+1]; // domain including boundary points
7
8      var X, XNew: [BigDomain] real = 0.0; // declare arrays:
9          // X stores approximate solution
10         // XNew stores the next solution
11
12     X[n+1, 1..n] = 1.0;          // Set south boundary values to 1.0
13
14     var iteration = 0,           // iteration counter
15         delta: real;            // measure of convergence
16
17     const north = (-1,0), south = (1,0), east = (0,1), west = (0,-1);
18
19     do {
20         // compute next approximation using Jacobi method and store in XNew
21         forall ij in ProblemSpace do
22             XNew(ij) = (X(ij+north) + X(ij+south) + X(ij+east) + X(ij+west)) / 4.0;
23
24         // compute difference between next and current approximations
25         delta = max reduce abs(XNew[ProblemSpace] - X[ProblemSpace]);
26
27         // update X with next approximation
28         X[ProblemSpace] = XNew[ProblemSpace];
29
30         // advance iteration counter
31         iteration += 1;
32     } while (delta > epsilon);
33
34 }
35

```

Figure 2: Chapel implementation of the Jacobi algorithm.

## 7 Practicalities

This section will give a brief account on our experiences with installing the different compilers and running the example codes. The installation of the compilers is in principle unproblematic: Chapel requires an up-to-date version of gcc/g++ and both Fortress and X10 require Java 1.5. However, on HPCx gcc is not supported, and in order to install and run the X10 compiler, the gcc compiler needs to be installed first. Java 1.5 is not the default installation on HPCx, but it is available on the system. Once the correct compilers were in places, all three language compilers were built without any problems.

The code examples that are provided with the language distributions largely also build correctly. A few compilation problems appeared in example for all languages, as would be expected with languages in the early stages of their development, but these were all hinted at in accompanying README files. Initially, the Chapel compiler (called *chpl*) had a bug and would not compile any of the provided examples, including the “Hello World” code. The *chpl* compiler had the use of *make* explicitly written into its source code, however on HPCx *make* is actually pointing to *gmake*. Changing the code to always invoke *gmake* solved the problem. The bug was reported to the Chapel team, who were very supportive and fixed the problem by introducing a command-line flag that allows a user to choose between *make* and *gmake* with no need to change the source code.

```

1 public class Jacobi extends x10Test {
2
3     const int N = 5;                // size of grid
4     const double epsilon = 0.0001; // convergence tolerance
5     const double epsilon2 = 0.000000001;
6
7     const region(:rank==2) RInner = [1:N, 1:N]; // region for grid points
8     const region(:rank==2) R = [0:N+1, 0:N+1]; // region including boundary
9                                           // points
10
11     // distribution of grid
12     const dist(:rank==2) D = (dist(:rank==2)) dist.factory.block(R);
13     const dist(:rank==2) DInner = D | RInner; // distribution for inner region of grid
14     const dist(:rank==2) DBoundary = D - RInner; // boundary region of grid
15
16     const int EXPECTED_ITERS = 97;
17     const double EXPECTED_ERR = 0.0018673382039402497;
18
19     final double[,] XNew = new double[D] (point p[i,j]){
20         return DBoundary.contains(p) ? (N-1)/2 : N*(i-1)+(j-1);
21     };
22
23     public boolean run() {
24         int iters = 0;
25         double err;
26         while (true) {
27             final double[:distribution==this.DInner] X = new double[DInner] (point [i,j]){
28                 return (XNew[i+1,j]+XNew[i-1,j]+XNew[i,j+1]+XNew[i,j-1])/4.0;
29             };
30             if ((err = ((XNew | this.DInner)-X).abs().sum()) < epsilon) break;
31             XNew.update(X);
32             iters++;
33         }
34         System.out.println("Error = "+err);
35         System.out.println("Iterations = "+iters);
36         return Math.abs(err-EXPECTED_ERR) < epsilon2 && iters == EXPECTED_ITERS;
37     }
38
39     public static void main(String[] args) {
40         new Jacobi().execute();
41     }
42 }

```

Figure 3: X10 implementation of the Jacobi algorithm.

The user communities are supported through forums and mailing lists, which prove to be very helpful as they are used by the language developers as well as new users. Any problems encountered when setting up the compilers or running the codes are thus quickly addressed.

## 8 Experiences and Conclusions

As mentioned at the beginning of this report, the main aims of the language design part of the HPCS programme are the development of highly productive parallel languages, which focus on performance, programmability, portability and robustness. The three languages presented here all use a similar approach in the design of parallelism, with partitioned global address spaces and multi-threading. However the languages differ from each other substantially in other aspects.

According to the HPCS aims, the new languages should be aimed at serial programmers and should be easy to learn for anyone who has programming experience. Fortress is clearly a language that focusses on the scientific community and scientists who need to develop code. This is underlined by the use of mathematical notation as a basis for the language's syntax and the static checking of units and dimensions. X10 on the other hand is aimed at Java programmers, using the same serial language subsets. Chapel is aimed at any code developers that have programmed in any modern (object-oriented) language.

The HPCS programme is a little over half-way through its lifetime and due to come to an end in the summer of 2010. The language specifications and compilers are therefore work in progress and their quality is variable. As the funding for SUN in this project was discontinued in 2006, Fortress is, as can be expected, in an earlier developmental stage as compared to X10 and Chapel. The Fortress compiler only implements a very small fraction of the language specification. Both the X10 and Chapel compiler are more advanced, however they also do not implement the full specifications. At the time of writing, parallel codes in any of the languages only run on shared memory nodes.

The productivity level of a language is not only defined by the scalability and performance of written code, but also by the time required to learn the language and develop the code. An experienced serial programmer, user to writing OO programs, would probably find X10 most accessible at first, due to its similarities with Java. However, when comparing X10's and Chapel's approaches to defining parallel task and data structures, Chapel has a more intuitive and direct way of describing these features than X10. Chapel allows a user to either define parallelism in a very simple way using built-in structures and definitions, possibly compromising on performance, or to use highly optimised, problem and machine specific structures and distributions, trying to get the best possible performance from code and machine. Chapel does not require a developer to have detailed knowledge about a machine architecture in order to quickly produce parallel programs with acceptable performance. Nevertheless, any such knowledge is beneficial and can be built into the code to increase performance. X10 on the other hand has a more complicated underlying structure and would be more difficult for a serial programmer, with no knowledge of HPC, to pick up. The documentation that is currently distributed with X10 is also not necessarily focussed at programmers that are novices in the field of HPC. Given that the syntax in Fortress is close to standard mathematical notation, the basics of this language should feel accessible to any developer.

Another criterium for a high productivity language is its support by tools and IDEs. Currently there exist Eclipse<sup>9</sup> plug-ins for both Fortress and X10. Thus far there is no support for Chapel; this is probably due to the fact that Chapel is not freely available at the present time.

To this date, a large amount of work has already gone into the development and specification of Fortress, Chapel and X10. The language designs are focussed on the programmability of next-generation high performance computing systems and the completion of

---

<sup>9</sup>Open Source IDE, <http://www.eclipse.org/>

their compiler implementations will show each language's capability of achieving their ultimate goal, which is the acceptance of the HPC community. Performance, ease of use, portability and safety will be the main factors that will decide whether or not these novel programming languages can challenge the position of MPI and OpenMP in the world of HPC. Those factors will not be able to be tested until stable, fully specification compliant releases of the compilers and supporting tools are available. 2010 is the deadline for X10 and Chapel – Sun will want to keep the development of Fortress at a similar pace. Whether or not any one of the three has the potential to become the next standard in HPC code development will become clearer in the next couple of years.

## References

- [1] High Productivity Computing Systems Website  
<http://www.highproductivity.org>
- [2] The Fortress Language Specification, Version 1.0  $\beta$ , March 2007.  
<http://research.sun.com/projects/plrg/fortress.pdf>
- [3] Report on the Experimental Language X10, Version 1.0.1, December 2006.  
<http://x10.sourceforge.net/docs/x10-101.pdf>
- [4] Chapel Language Specification 0.750, 2007.  
<http://chapel.cs.washington.edu/spec-0.750.pdf>
- [5] B.L. Chamberlain, D. Callahan, H.P. Zima. Parallel Programmability and the Chapel Language. In *International Journal of High Performance Computing Applications*, 21(3): 291-312, August 2007.  
<http://www.highproductivity.org/HPPLM/final-chamberlain.pdf>