Science & Technology Facilities Council
Rutherford Appleton Laboratory

Technical Report

RAL-TR-2007-012

02/08/07

New Languages for High Performance, High Productivity Computing

J.V. Ashby
Computational Science and Engineering Department
STFC Rutherford Appleton Laboratory
J.V.Ashby@rl.ac.uk

Abstract

The High Productivity Computing Systems (HPCS) project aims to create a new generation of high end programming environments, software tools and computer architectures. One arm of the project is developing new languages to support the high productivity computational environment envisaged. Three languages, Fortress, Chapel and X10, were defined in the first phase of the project, and the latter two are being further developed in phase 2. This report gives an overview of the features of the three languages, followed by amore detailed discussion of the support they offer for parallel computing. All three possess the basic attributes one might expect of a parallel language. task parallelism, data distribution and synchronisation, but express them in different ways and with different levels of programmer control.

1. Background

The development of scientific computing has been accompanied over many decades by the development of programming languages. The first language to remove the programmer from the mind-bending task of working out how to perform an algorithm in machine code was Fortran, first announced in 1956 [1], rapidly followed by Algol (1960, revised in 1968) [2], Pascal (1970) [3], C (early 1970's) [4] and ADA (1983) [5]. Concurrently languages were developed for other application domains such as COBOL (1960) [6] for business programming, prolog (1975) [7] and Lisp (1959) [8] for Artificial Intelligence, and PL/1 (1964) [9] for data processing (although it could be argued that PL/1 is one of the broadest based languages, covering application areas from systems to scientific and commercial programming). The next major development was the advent of object oriented languages such as C++ (1985) [10] , Java (originally designed for embedded systems, but quickly applied more widely, 1995) [11] and smalltalk-80 (1980) [12]. With these, the emphasis was placed much more on the design philosophy of a program, and the languages were intended to be general purpose. Scripting languages had long been a feature of computing – Basic [13] was already an interpreted language in 1975, and most operating systems incorporated a scripting mechanism, but in recent years a multiplicity of scripting languages has emerged, particularly in conjunction with web-based applications, including VBscript [14], javascript [15], perl [16], python [17] and ruby [18].

With the advent of parallel and vector computing, an early goal was the production of automatic parallelizing compilers. This proved difficult in the framework set by existing languages, though automatic vectorization was successful. Several attempts were made to add seamlessly to a language by means of directives that a compiler was free to act on or not. Typical of these is High Performance Fortran (HPF) [19], which included directives that could be treated as comments by a compiler, or could give suggestions for ways to exploit parallelism within the code. However, it was widely felt that such directive based parallelization was insufficiently rich to describe the algorithms that were emerging, and the message passing paradigm became established. Here a library of routines is supplied (MPI [20], PVM [21] and P4 [22] are examples) by means of which processes can communicate and cooperate. The functionality of the library is defined outside any specific programming language, and a language binding is supplied as an API. Recently a return to incorporating parallelism within programming languages has occurred, with CAF, UPC and Titanium [23] looking for the minimal set of additions to existing languages (Fortran95, C and Java respectively) which will support parallelization based on the Partitioned Global Address Space model. In these languages the communications and synchronization are largely abstracted out, leaving the programmer writing code that could work equally well on shared or distributed memory architectures.

Another strand of language development that should be noted is the development of functional languages and systems such as Matlab [24] or Mathematica [25], which can be regarded as languages for rapid prototyping. They are very useful for testing out algorithmic ideas, but generally are restricted when it comes to production code, either by being too slow or by being limited in the size of problems that can be tackled. However, they are useful, and there would be much interest in a language that provided a seamless transition from the rapid prototype to a highly optimized production version with full support for parallelism.

2. The Challenge

In 2002 the US Defense Advanced Research Projects Agency (DARPA) initiated a programme called HPCS, High Productivity Computing Systems [26]. The mission of this program is to realize a new vision of high productivity computing systems by the creation of new generations of high end programming environments, software tools, architectures, and hardware components. This will bridge the gap between existing supercomputing capacity and the promise of quantum computing [26]. Attention is focussed on the four main attributes of Performance, Programmability, Portability and Robustness. The programme was designed in three phases: in Phase 1 for twelve months there would be an assessment of current and emerging technology, the development of fresh solutions and the generation of new productivity metrics; Phase 2 would be 36 months of research and prototype development; finally phase 3 would take the best ideas and develop them into full-scale products over 48 months. This report is being written at the beginning of phase 3.

Developments under this programme are proceeding on several fronts, hardware, software, systems characterisation, etc. In particular HPCS identified a need for a new language which would support the high productivity environment it was trying to create. It should be noted here that Performance is only one of four criteria the new language can be judged against – the aim was not simply to produce a language that ran programs as fast as possible, but also one in which programming safe reliable code is easy and swift. A good analysis of the challenges involved can be found in [27]

To this end DARPA commissioned three major computer vendors to produce prototype next generation systems including new programming languages. Sun Microsystems [28] have developed new modes of chip-to-chip communication through novel VLSI fabrication techniques, photonic interconnect, smart storage systems and the Fortress programming language, which aims “to do for Fortran what Java did for C”. Sun were not, however, selected to participate in phase 3.

Cray developed a system called CASCADE [29] based on a heterogeneous processing model where a combination of system software and compiler/execution environment can map applications onto available processor hardware. The system is likely to be based on AMD multicore Opteron processors and to exploit third party tools such as the Portland Group compilers, Cluster File Systems' file systems and DataDirect Networks storage technology. For a language solution Cray produced Chapel in collaboration with CalTech/JPL. Its aim was to provide a higher level of expression than current parallel languages and to improve the separation between algorithmic expression and data structure implementation.

The third partner who, together with Cray, are involved in phase 3, is IBM. Their project, codenamed PERCS (Productive, Easy-to-use, Reliable Computing System) [30] will be based on Power7 chips and IBM's own General Parallel File System (GPFS). The X10 language offered by IBM is arguably the least radical of the three developed in phase 2, being largely a re-engineering of Java for parallelism.

In the next section we shall look briefly at the main features of the three languages, Fortress, Chapel and X10. Following that we shall look in more detail at the support they offer for parallel computing and finally make some concluding remarks.

3. Brief attributes of the new languages

Fortress

The Fortress language is being developed by Sun Microsystems [28], and is intended to be a growable, general purpose language for robust, high performance software with high programmability. This is a large goal, and the language is correspondingly large and complex.

Fortress as a name is intended to invoke the idea of a "secure Fortran", combining the ideals of Fortran for high performance computation with the abstraction and type safety inherent in modern languages such as C++ and Java. It is, however, not an evolution of Fortran. Rather it is a language being developed from scratch, though informed by experiences with other languages including Java, NextGen [31], Scala [32], Eiffel [33], Self [34], Standard ML [35], Objective Caml [36], Haskell [37] and Scheme [38].

The basic concepts in Fortress are objects and traits [39]. Objects are familiar from OO programming, they are composed of fields and methods. Traits are named sets of methods, which provide a method of inheritance that is said to be better than conventional class inheritance. Within a trait a method may be abstract (consisting only of headers, similar to an interface declaration in Fortran) or concrete, carrying with it code defining the action it embodies.

In a radical departure from most other languages previously used by computational scientists, Fortress is not restricted to being written in ASCII characters. Unicode characters can be used (giving access for example to the Greek alphabet and many mathematical symbols), as well as sub- and superscripting. Types are inferred where possible and operators can be overloaded. These points allow programmers to write Fortress in something that looks a lot like standard mathematical notation. In this Fortress seeks to ease the transition from a rapid prototype such as might be written in Matlab or Mathematica to a high performance application.

As well as possessing values, object in Fortress can also be associated with physical units such as metres or seconds, and dimensions such as length or time. Dimensional checking is carried out statically (essentially at compile time) and errors reported. Consistent and sensible use of this facility should reduce bugs and improve the correctness of code.

Aggregate expressions are Fortress' method for using many kinds of collections of data such as tuples, arrays, matrices, vectors, maps, sets and lists. Arrays are defined by writing their elements enclosed in square brackets `[]`. Two dimensional arrays have rows separated by new lines or by semicolons. In three dimensional arrays pairs of semicolons separate the two-dimensional sub -arrays and so on. Sets are enclosed in braces (curly brackets `{}`) while list elements are enclosed in angle brackets (written in ASCII as `<|` and `|>`).

A Fortress program is composed of nested blocks of code. The entire program is a block, and each component is a sub-block. Some expressions are also blocks or have blocks as part of themselves. For example a while expression is a block (the part that is tested) and its body (the part that is executed) is another block. Blocks may be sufficiently independent that they can be executed in parallel or in a separate thread. This is one example of Fortress' implicit support for parallelism. Others are that tuple expressions may be evaluated in several implicit threads, and that `for` loops and similar constructs are parallel by default (programmers can override this by use of the sequential qualifier). More details

of the support for parallelism in Fortress are given in section 5.

In January 2007 Sun released a JVM-based interpreter for a core set of the Fortress language [40]. There is some speculation that this may have been prompted by an announcement that the project would not be proceeding to the next stage of funding, leading to Sun seeking to establish the language as a *de facto* standard.

Chapel

Chapel is the language being developed by Cray Inc in association with CalTech/JPL as part of their overall contribution to the HPCS project, Cascade. Chapel itself is the Cascade High Productivity Language [41].

Chapel aims to bridge algorithm development and production deployment in the domain of high-performance parallel computing. Thus while it incorporates many of the features of modern programming practice such as object orientation, some compromises are made in order to achieve high performance.

Two basic techniques underpin the approach of Chapel: Locality Aware Multi-Threading and Generic Programming. Locality Aware Multi-Threading is an extension of the PGAS (Program Global Address Space) model expressing the relationship between computations on a particular process and the data they access. Generic Programming addresses the issue of code re-use by providing constructs which express the abstract qualities of data-structures in such a way that algorithms can then be expressed to accept any data-structures which posses the required qualities. This means, for example, that an algorithm which included multiplication would "automatically" be defined for integers, reals, complex numbers, quaternions, matrices, but not for strings, say.

As a basic language, Chapel is an amalgam of the most successful elements of several important high-level languages, notably Fortran90/95, C and C++. Programmers familiar with those languages should be able to follow the Chapel programming model, though some of the terms by which concepts are called may change (and there may be some false friends – terms which appear familiar but are actually expressing a slightly different concept). The language is object-oriented, with inheritance and overloading supported. Structured types (Fortran derived types, C structs) can be defined together with constructors and bound functions for them. The `use` statement (which here as in Fortran incorporates code and variables from a module) can also be used to control access to fields in a structured type. Union types are also supported – variables which can hold one of several types of variable (e.g. a floating point number or an integer). To access these safely a `typeselect` control statement is provided.

A further important high-level type is the sequence. This is used to provide iteration over sets in an abstract fashion (thus leaving details of implementation to the compiler or to the system). A sequence is a list of expressions (all of which must have the same type). Arithmetic sequences can be constructed using a special operator, `<expr1> .. <expr2>` and subsequences can be constructed using the by operator. Sequences can be used to control for loops and similar iterators. For example if `X` is a sequence, then to square all its members we could use the expression `[y in X] y*y.`

Programmers in most high-level languages are familiar with the concept of aggregate data collections such as arrays. Chapel extends this idea by introducing high-level objects called domains. These are not the aggregate data collections themselves, but are descriptions of how such collections are addressed. A conventional Fortran or C array is addressed by a set of integers, as many as the rank of the array requires, which must be within certain limits. Thus its domain is specified as so many bounded integers. In Chapel, domains are not restricted to integers but can be addressed by strings, booleans, enumeration types, or any other scalar type. For many of these it does not make sense for bounds to be provided (in what way would a string index run from '`nickel`' to '`silver`'?) and so the elements of domains defined by these types varies as the program runs. Methods for adding and deleting indices to a domain are provided, as well as methods for querying if an index is already defined.

X10

X10 is arguably the smallest and least ambitious of the three languages. It is described in its reference manual [42] as "(generic) Java less concurrency, arrays and built-in types, plus *places, activities, clocks,* (distributed, multi-dimensional) *arrays* and *value* types." Thus it represents an attempt to migrate an existing language to the field of High-Productivity, High-Performance computing in the same way as Co-Array Fortran, UPC and Titanium. The difference is that whereas those three languages use a Partitioned Global Address Space model to express parallelism, X10 extends this to a Globally Asynchronous, Locally Synchronous (GALS) model.

The basic elements of X10 follow the Object Oriented paradigm of Fortress and Chapel. Generic interfaces and classes are the building blocks of X10 programs. Scalar classes possess fields, methods and inner types, they can inherit attributes though sub-classing another class. Classes may be of reference type or of value type. Value classes are less flexible than reference classes, but can be implemented efficiently. They are classes the fields of which are all final, that is they cannot be assigned to more than once and must be assigned before use. Unlike in Java it is possible to explicitly add a null value to the set of values a type may assume through the nullable prefix type constructor.

Array class declarations are not supported by X10, which means that a user cannot define new array class types. Arrays are created as instances of array types through array type constructors.

X10 does not have a dereference operation, and does not support pointer arithmetic. This renders code much safer. X10 is said to be type safe (locations are guaranteed to contain legitimate values for the type that refers to them), memory safe (accesses are bound checked dynamically and automatically) and pointer safe (non-nullable values cannot throw a null pointer exception). In addition they are place safe (a place being X10's nomenclature to describe the locality of data and activity) and if they use only clocks and unconditional atomic sections for synchronization, they are guaranteed not to deadlock.

4. Parallelism in the new languages

Fortress

As we said in section 4, Fortress supports parallelism implicitly in a number of its syntactical features. Fortress defines locality for threads and data through a region. This is an abstract representation of a section of the machine in which the thread or data resides. In a distributed memory multi-processor each region might be a single processor and its associated memory, The distribution of data across regions is defined by default by Fortress, depending on the size and shape of the array or other aggregate and of the size and configuration of the machine. Programmers can also determine the distribution of data through the distribution trait. Several built-in distributions are provided such as `sequential` (blocks of contiguous address space allocated to regions), `par` where one address is allocated to each region in turn, starting again at the beginning when the number of regions is exhausted (like dealing a deck of cards), `blocked` where the space is divided into roughly equal chunks, and `subdivided` where the address space is recursively bisected.

Synchronisation and data locking in Fortress is achieved through the use of atomic code blocks and expressions. An atomic expression is a set of instructions which must either not have been started or have finished before another thread can access the data referenced therein. The example given in the language definition is the following block of code:

```
do
     x = 0
     y = 0
     z:ZZ := 0
     (atomic do
          x += 1
          y += 1
     end,
     z := x+y)
     z
end
```

The first and last lines enclose a block of code which returns a value `z`. The 2nd to 4th lines initialise `x`, `y` and `z`, the last of which is defined as an integer (member of the set `ZZ` which would be rendered as a double-struck Z). Then the parentheses enclose a duple expression which is executed in two parallel threads by default. However, because the first element of the duple is an atomic expression incrementing both `x` and `y`, the thread evaluating the second element is blocked from accessing them while it is running. Thus the whole block returns either 0 or 2. However, which of the two it will return is non-determined.

Chapel

The support for parallelism and distribution in Chapel is somewhat richer than in Fortress, though it is predicated on a similar model. Again several constructs are parallel by default; these include operations of arrays, domains and sequences. In addition there are three control constructs which explicitly introduce parallelism into a program. These are `forall`, `cobegin` and `begin`.

The `forall` loop is a version of the `for` loop which allows concurrent execution of the loop body. The execution is not guaranteed to be concurrent, merely permitted to be. Whether it is or not will be decided by the compiler and runtime system based on data dependency and resource availability. A simple example of the `forall` statement is in the copying of one vector into another:

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

where the programmer has asserted that `a` and `b` are sufficiently independent that the assignments may be carried out concurrently. An alternative syntax for this loop would be:

```
[i in 1.N] a(i) = b(i);
```

The keyword ordered can be prepended to a `forall` statement to constrain the concurrency, Thus:

```
ordered forall i in walk(root) do
     something(i);
iterator walk(n: node) {
yield n;
forall c in 0..n.numOfChildren{
     yield n.child(c);
     }
}
```

The iterator walk effectively orders the nodes such that no child may precede its parent node, producing a breadth-first tree walk.

The alternative syntax for the `forall` statement can also be used to evaluate expressions concurrently. Thus:

```
[i in S] f(i);
```

would evaluate `f(i)` where `i` ranged over `S`, possibly spawning one process or thread for each value of `i`, and return the results in a sequence. Placing the `ordered` keyword in front of this suppresses parallelism in the iteration over `S` and forces the execution to occur in sequence order. It does not, however, suppress any parallelism there may be in the evaluation of `f(i)`.

The `cobegin` statement is an indication that the statements in the block it introduces can be executed concurrently. An example would be:

```
cobegin {
     a = b;
     [i in 1..N] c(i) = d(i);
}
```

where the assignment of `b` to `a` can be carried out independently of the (multiple, concurrent) assignments of `d` to `c`. All the concurrent computations must finish before control can continue beyond the block. Control may not be transferred into or out of the `cobegin`'s block statement. The same is true of the `forall` statement.

In contrast, the `begin` statement spawns off a thread or process to execute its associated computation, while control in the spawning thread or process continues. This could be used, for example, to separate off I/O from the working body of the code.

In the same way that the `ordered` keyword can control the degree of parallelism in a code, so can the `serial` statement. In this case a test is given, and if it evaluates to true, implicit parallelism (such as in a `forall` loop) is suppressed. Thus in:

```
serial depth >3 forall i in S {
     do some work
}
```

the `forall` loop will be executed concurrently if `depth` is less than or equal to 3, but serially when `depth` exceeds 3.

A special class of variable is defined in Chapel to support parallel computation. Synchronization variables are used, as the name suggests, to coordinate and synchronize computations. Reference and assignment of these variables controls the order of execution. There are two forms of synchronization variables: `single` and `sync` variables.

A `single` variable can only be assigned while it is active (in scope). A reference to a `single` variable before it has been assigned acts to suspend execution of the referencing process or thread until assignment has been made. Any variable defined within a `cobegin` statement is implicitly single.

`Sync` variables are a generalisation of single variables that permit multiple assignments. `Sync` variables may be in one of two states, full or empty. Attempting to reference an empty variable will suspend execution of the referencing computation until the `sync` variable becomes full (by being assigned). Reading from the variable will change it back to being full. The variable is like a message basket which can only hold one message at a time. If the basket is empty a process must wait until it becomes full to take out a message, if it is full a process must wait until it has been emptied before delivering the next message.

It is possible to make a block of code `atomic` – that is from the point of view of the rest of the program the block appears to execute as a single order, and no changes to its variables are apparent until it has finished.

Thus far we have addressed how Chapel exploits parallelism automatically in a task based fashion. The model has been of a shared memory machine generating threads or processes to handle executing multiple statements or blocks of code simultaneously. Using Locales and Distributions, it is possible to associate data and computations with specific locales (and hence, implicitly, with each other).

A locale is an abstraction of a processor or node in a parallel computer. Both data and computation can be associated with locales, thus the programmer can be in control of both data and work placement. There is a predefined constant array of locales which is instantiated at runtime with a configurable size. The program initially runs on the locale described by the first element of this array. Computations are directed towards a specific locale or set of locales by use of the on statement. This has the form:

```
on <expression> do <statement>
```

or

```
on <expression> <block>
```

Expression should be either a locale or a variable. In the latter case the computation is run on the locale where the variable is located. Thus:

```
forall i in D do on A(i) {
some work
}
```

will execute the `forall` loop in such a way that the `i`-th instance executes on the locale where `A(i)` is located.

The location of data is controlled by distributions. These map domain indices to locales. Iteration over a distributed domain or array then implicitly executes the computation on the associated locales. There are a set of standard distributions including Block, Cyclic, BlockCyclic and Cut. The ability for the user to define distributions is planned, but this facility is still in development.

Finally the Chapel language provides built-in reductions and scans, and the potential for users to define more reductions and scans. This latter is by a class definition implementing a structural interface.

X10

X10's name for the entity which localises data and computation is the *place*. Activities running in a particular place may access the data associated with it at maximum speed, access to data in other places may take much longer. There is a built-in class for the place of which all places are instances. Currently this class is not extensible. Various attributes of this class define such things as the maximum number of places available and the set of all places active.

In X10 a segment of the computation which runs concurrently with others is called an *activity*. Activities are spawned using the `async` statement:

```
async <place> <statement>
```

where place may be blank or may be an expression which evaluates to a single place. If an object is given instead, the location of that object (`obj.location`) is used instead. One potential difficulty with this comes when using a distributed object, since `a[i]` will be evaluated as `a[i].location`, which may not be the same as `a.distribution[i]`. The programmer must be aware of this and use appropriate expressions.

A distribution maps a region (a set of indices describing locations in memory or pseudo-memory) to a set of places. The distribution `D` has associated attributes `D.region` (its underlying region) and `D.places`, the set of places forming the range of `D`. If `D` is applied to a point `p` in the region (by writing `D[p]`) it returns the place to which `p` maps. As with the other languages, a set of standard distributions is supplied including the constant distribution in which every point n the region maps to the same place, block, cyclic and block cyclic distributions, and an arbitrary distribution. Distributions can be restricted over domains or over ranges (places). The way in which an implementation does this may vary for the particular needs of an architecture.

As with Fortress and Chapel, X10 supports the notion of *atomic* blocks of code to coordinate and synchronize threads. An unconditional atomic block is a statement or block of statements preceded by the keyword atomic. This is executed as a single step, and all concurrent activities in the same place are suspended. Since there is no impact on activity in other places, this is a non-blocking form of synchronization. A conditional atomic block has the form:

```
when (boolean expression) {
     statement block
}
or (other boolean expression) {
     other statement block
}
.
.
.
```

The `or` clause is optional and there may be several of them. When the program reaches a conditional block it will suspend execution until one of the boolean expressions (called guards) evaluates to true. At that point the first block whose guard is true will be executed. With this construct it is possible to implement locks and waits.

Barrier style synchronization is implemented using a generalization of barriers called *clocks*, which allow the registration and deregistration of dynamically created activities. They are designed to avoid deadlocks and race conditions.

5. Current Status and Roadmaps

Fortress

As stated above, in early 2007 Sun released a Java based interpreter for Fortress and made the project open-source. This coincided with the HPCS project deciding not to continue with Fortress. Development of the interpreter and libraries is continuing, both at Sun and in teams outside the company. It is expected that work will begin on a compiler by 2008 with a complete 1.0 version of a non-optimising implementation of Fortress by late 2008. [43]

Chapel

Chapel has been exercised on the HPC Challenge suite of benchmarks [44]. There is a prototype compiler (not yet widely available outside the project) which currently does not support distributed architectures, nor all the serial features of the language [45]. It works on a single locale, and supports multiple tasks within a locale, so can be used for multicore processors or for a shared memory node. Task parallelism is supported through multi-threading. Data parallel structures are implements, but do not result in parallelism within the locale. This and distributed systems are the two main goals for the

next phase of implementation. The performance of this compiler is good, and it has been released to HPCS mission partners and a handful of users who have expressed interest in evaluating it and providing feedback to assist the development path. The next release on similar terms is due in December 2007 with a public release towards the end of 2008. It is expected that it will take a while after that for performance to catch up with hand-coded MPI.

While Chapel is being developed in the framework of a specific Cray hardware configuration, the focus is on implementing it for generic parallel architectures, and the current releases are targeted at Linux clusters. [46]

X10

The latest news on X10 [42] states that the development team have a well-defined langauage with formal semantics which have been implemented in a parallel, single VM compiler (thus handling shared memory, but not distributed memory architectures) which has been exercised on several benchmarks and applications. Further development of tools and a distributed memory compiler are expected, though there are no timescales currently available.

6. Conclusions

The three approaches to defining a new language for High Productivity, High Performance computing show a number of similarities and several differences. They have all taken a broadly similar model for parallel computing, in particular adopting a unified concept of locality for both data and computation, with the use of atomic constructs for synchronisation. Chapel has gone further and introduced special synchronization variables which also assist memory consistency.

It should be remembered that the aim of the HPCS programme is high programmer productivity, and arguably both Fortress and X10 have concentrated on this in different ways, leaving high performance in second place. Since programming for performance is likely to be a major concern for the computational science community, the value of the new languages will in large part be determined by the quality of the compilers and associated tools they provide. Existing computational scientists have a large investment in dusty deck code, and evolution will always be easier than revolution. Few projects can afford the luxury of putting a large piece of software on one side and re-writing it in a new language. At the very least, well-defined interoperability is required, with such matters as the interface by which libraries of old code (written in Fortran, C or C++) can be called. None of the languages currently appears to tackle this problem.

Fortress is clearly an ambitious project, trying to squeeze the best of many different languages into one. The language suffers from being very large as a result of this, and the support for parallelism relies heavily on the compiler knowing best. The theoretical basis of the language is strong (as it is for X10), but a language that can be used by scientists, and not just by computer scientists, requires a pragmatic foundation as well. X10 has taken a more evolutionary (some might say timid) path, seeking to amend Java in as minimal as possible a fashion to achieve the HPCS goals. In doing so it has also kept its parallel constructs to a minimum and ultimately performance would rely on clever optimization and automatic parallelization.

Chapel seems to offer the best way forward for experienced HPC programmers who wish to exploit its richness and retain the ability to squeeze the last drop of performance out of an algorithm. Its parallel constructs are syntactically rich and, as one might expect from Cray, well-suited to use by knowledgeable practitioners. On the debit side, however, it is also a very large language and many of the ways in which it is described in the language manual are esoteric. Whichever language emerges as the preferred choice, there will be an urgent need for readable introductory textbooks which remove the mysterious elements from the formal specification.

Finally, it is worth raising the question of whether one language can hope to cover such a large set of requirements. Might it not be better (and in the spirit of such contemporary paradigms as agile programming) to produce several small programming languages, each of which is excellent at a small sector of the target programming space, with well-defined methods for inter-operability? It is possible that this could be done by modularising the three proposed languages. The advantage is that it becomes clear which language (or part of the language) a programmer needs to know to perform a specific task, reducing the time spent on the learning curve and increasing programmer productivity in a way the current offerings do not.

Acknowledgments

I would like to thank M. Ashworth and R.J. Blake for constructive comments.

References

1. Metcalf M. and Reid J.K Fortran95 Explained, Oxford Science Publications (1998)
2. Bauer F.L. An Introduction to Algol, Prentice Hall 1964
3. Morton J.K. An Introduction to Pascal, Business Educational Publishers Ltd 1993
4. Kernighan B.W. And Ritchie D.M., The C Programming Language (2nd Edition) Prentice Hall 1988
5. Barnes J. Programming in Ada 2005, Addison Wesley (2006)
6. Shelly G.B. And Cashman T.J., Structured COBOL, Boyd and Fraser (1984)
7. Bratko I., Prolog Programming for Artificial Intelligence, Longman (2001)
8. McCarthy J., The List 1.5 Programmer's Manual, MIT Press (1963)
9. Smedley D., Programming the PL/1 Way, McGraw-Hill (1982)
10. Stroustrup B., The C++ Programming Language, Addison Wesley (2000)
11. Flanagan D., Java in a Nutshell, O'Reily (2005)
12. Korienek G. and Wrensch T., A Quick Trip to ObjectLand: Object-Oriented Programming with Smalltalk/V, Prentice Hall (1993)
13. Clark J.F., BASIC, a Structured Approach, South-Western (1989)
14. Wilson E., Microsoft VBScript Step by Step, Microsoft Press (2006)
15. Flanagan D., Javascript, the Definitive Guide, O'Reilly (2006)
16. Wall, L., Christiansen T. and Orwant J., Programming Perl, O'Reilly (2000)
17. Lutz M., Programming Python, O'Reilly (2006)
18. Carlson R. and Richardson L., Ruby Cookbook, O'Reilly (2006)
19. Koelbel C.H., Loveman D.B., Schreiber R.S., Steele G.L. And Zosel M.E., The High Performance Fortran Handbook, MIT Press (1994)
20. Gropp W., Lusk E. and Skjellum A., Using MPI: Portable Parallel Programming with the Message-passing Interface, MIT Press (2000)
21. Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R. and Sunderam V.S., PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press (1995)
22. Butler R. and Lusk E., User's Guide to the p4 Parallel Programming System, http://www-fp.mcs.anl.gov/~lusk/p4/p4-manual/p4.html
23. Ashby J.V. Novel Parallel Languages for Scientific Computing – a comparison of Co-Array Fortran, Unified Parallel C and Titanium, Rutherford Appleton Laboratory Technical report RAL-TR 2005-015 (2005)
24. Higham D.J. and Higham N.J., Matlab Guide, S.I.A.M. (2005)

25. Wolfram S., The Mathematica Book, Wolfram Media Inc (2004)

26. http://www.darpa.mil/ipto/programs/hpcs/index.htm High Productivity Computing Systems (HPCS)

27. Snir M., Programing Languages for HPC – Is There Life After MPI? http://www.cs.uiuc.edu/homes/snir/PDF/Programming%20Languages%20for%20HPC%20short.pdf

28. http://research.sun.com/spotlight/2006/2006-04-07_Sun_on_HPCS.html Vildibill, M. Sun on HPCS: Changing the Productivity Game.

29. http://www.cray.com/products/programs/cascade/index.html Cascade – HPC Technology Initiatives

30. http://domino.research.ibm.com/comm/pr.nsf/pages/news.20030710_darpa.html

31. Cartwright R. and Steele G., Compatible genericity with run-time types for the Java Programming Language. In OOPSLA (1998)

32. Odersky M., Altherr P., Cremet V., Emir B., Micheloud S., Mihaylov N., Schinx M., Stenman E. and Zenger M., The Scala Language Specification. http://scala.epfl.ch/docu/files/ScalaReference.pdf

33. Meyer B., Object Oriented Software Construction, Prentice Hall (1988)

34. Agesen O., Bak L., Chambers C., Chang B-W., Hölzle U., Maloney J., Smith R.B., Ungar D. and Wolczko M., The Self Programmer's Reference Manual, http://research.sun.com/self/release_4.0/Self4.0/manuals/Self-4.1-Pgmers-Ref.pdf

35. Milner R., Tofte M., Harper R. and MacQueen D., The Definition of Standard ML, MIT Press (1997)

36. Leroy X., Doligez D., Garrigue J., Rémy D. and Vouillon J., The Objective Caml System, release 3.08, http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf

37. Peyton-Jones S., Haskell 98 Language and Libraries, Cambridge University Press (2003)

38. Kelsey R., Clinger W. and Rees J., Revised report on the algorithmic language Scheme, ACM SIGPLAN Notices, **33(9)**, 26-78 (1998)

39. Allen E., Chase D., Hallett J., Luchango V., Maessen J-W., Ryu S., Steele G.L. and Tobin-Hochstadt S., The Fortress Language Specification, Version 1.0a http://research.sun.com/projects/plrg/Publications/fortress1.0alpha.pdf

40. http://fortress.sunsource.net/ Fortress Project home. This page contains links to many reports on Fortress.

41. http://chapel.cs.washington.edu/ Chapel – the Cascade High Productivity Language. This page contains links to many reports on Chapel.

42. http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html The X10 Programming Language. This page contains a link to a presentation on X10 by V. Sarkar.

43. Steele, G. Private Communication

44.HPCC Benchmarks http://icl.cs.utk.edu/hpcc/

45.Chamberlain, B., Chapel; Global Benchmarks andStatus Update. Delivered at Cray User Group meeting, May 2007

46.Chamberlain, B. Private Communication