The Cascade High Productivity Language*

David Callahan†, Bradford L. Chamberlain†, and Hans P. Zima‡
†*Cray Inc., Seattle WA, USA, {david, bradc}@cray.com*
‡*JPL, Pasadena CA, USA, zima@jpl.nasa.gov*

Abstract

The strong focus of recent High End Computing efforts on performance has resulted in a low-level parallel programming paradigm characterized by explicit control over message-passing in the framework of a fragmented programming model. In such a model, object code performance is achieved at the expense of productivity, conciseness, and clarity.

This paper describes the design of Chapel, the Cascade High Productivity Language, which is being developed in the DARPA-funded HPCS project Cascade *led by Cray Inc. Chapel pushes the state-of-the-art in languages for HEC system programming by focusing on productivity, in particular by combining the goal of highest possible object code performance with that of programmability offered by a high-level user interface. The design of Chapel is guided by four key areas of language technology: multithreading, locality-awareness, object-orientation, and generic programming. The Cascade architecture, which is being developed in parallel with the language, provides key architectural support for its efficient implementation.*

1. Introduction

The almost exclusive focus of current High End Computing (HEC) systems on performance has led to a dominating programming paradigm characterized by a localized view of the computation combined with explicit control over message passing, as exemplified by a combination of Fortran or C/C++ with MPI. Such a *fragmented memory model* provides the programmer with full control over data distribution and communication, at the expense of productivity, conciseness, and clarity. Thus, quite in contrast to the successful emergence of high-level sequential languages in the 1950s, parallel programming for HEC systems is conducted today using an assembly language-like paradigm, a consequence of the difficulty of obtaining performance in any other way.

Numerous projects over the past decade have tried to improve this situation by proposing higher-level languages that provide a global view of the computation and enhance programmer productivity, such as High Performance Fortran (HPF) and its variants. However, these languages were not accepted by a broad user community, mainly for the fact that the generated object code could not compete with the performance of "hand-coded" programs using MPI or other message passing libraries. A major reason for this shortcoming is the inadequate support for scalable and efficient parallel processing in many conventional architectures combined with a lack of language expressivity and weaknesses in compilers and runtime systems.

In this paper we discuss the design of a new language called *Chapel*—the *Cascade High Productivity Language*—in the context of an architecture development targeting a Petaflops computing system. *Cascade* is a project in the DARPA-funded *High Productivity Computing Systems (HPCS)* program led by Cray Inc., with the California Institute of Technology, NASA's Jet Propulsion Laboratory (JPL), and Stanford and Notre Dame Universities as partners.

Chapel pushes the state-of-the-art in programming for HEC systems by focusing on *productivity*. In particular Chapel combines the goal of highest possible object code performance with that of *programmability* by supporting a high level interface resulting in shorter time-to-solution and reduced application development cost. The design of Chapel is guided by four key areas of programming language technology: multithreading, locality-awareness, object-orientation, and generic programming.

1) *Multithreaded parallel programming* in the style of Multilisp, Split-C, or Cilk, supports fine-grain parallelism and resource virtualization so that each software component can express the concurrency that is natural to it. This facilitates latency tolerance, allows for automatic management of pro-

* This material is based upon work supported by the Defense Advanced Research Projects Agency under its Contract No. NBCH3039003. The research described in this paper was partially carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

cessors, and provides a clean interface for crafting composable software components.

2) *Locality-aware programming* in the style of HPF and ZPL provides distribution of shared data structures without requiring a fragmentation of control structure. The programmer reasons about load-balance and locality by specifying the placement of data objects and threads.

3) *Object-oriented programming* helps in managing complexity by separating common function from specific implementation to facilitate reuse.

4) *Generic programming and type-inference* simplify the type systems presented to users. High-performance computing requires type systems to provide data structure details that allow for efficient implementation. Generic programming avoids the need for explicit specification of such details when they can be inferred from the source or from specialization of program templates.

This paper is structured as follows. Section 2 will discuss languages for scientific parallel programming developed during the past decade. The main contribution of the paper is a description of the major design elements of Chapel in Section 3. This will be followed by an overview of the Cascade system architecture in Section 4. The paper concludes with a discussion of open issues and an outlook to future work in Section 5.

2. Languages for Scientific Parallel Programming

With the emergence of distributed-memory machines in the 1980s the issue of a suitable programming paradigm for these architectures, in particular for controlling the tradeoff between locality and parallelism, became important. The earliest (and still dominant) approach is represented by the *fragmented programming model*: data structures and program state are partitioned into segments explicitly associated with regions of physical memory that are *local* to a processor (or a small SMP); control structures have to be partitioned correspondingly. Accessing non-local state is expensive. The overall responsibility for the management of data, work, and communication is with the programmer. The most popular versions of this explicitly parallel approach today use a combination of C, C++, or Fortran with MPI.

It soon became clear that a higher-level approach to parallel programming was desirable and feasible, based on data-parallel languages and the *Single-Program-Multiple-Data (SPMD) paradigm*, with a single conceptual thread of control coupled with user-specified annotations for data distribution, alignment, and data/thread affinity. For such languages, many low-level details can be left to the compiler and runtime system. *High Performance Fortran (HPF)* became the trademark for a class of languages and related compilation and runtime system efforts that span more than a decade. Some of the key developments leading to HPF include the Kali language and compiler [13], the SUPERB restructuring system [23], and the Fortran D [9] and Vienna Fortran [5, 24] languages, both of which proposed high-level language extensions for parallel programming in Fortran77. HPF-1 [10], completed in 1993, was welcomed by many in the user community but was soon recognized as being too constrained in its data distribution features, resulting in performance drawbacks for important classes of applications. HPF-2 [11], the current *de facto* standard, and HPF+ [4] both extended the power of the distribution mechanism to accommodate dynamic and irregular applications [14]; HPF+ took the additional step of providing high-level access to low-level mechanisms such as the management of communication schedules and "halos" in order to allow user control of communication. The importance of these additional features is highlighted by the success of JA-HPF, the Japanese version of HPF derived from HPF+, which recently achieved a performance of 12.5 Teraflops for a plasma code on the Earth Simulator [18].

As HPF decreased in popularity, a number of languages rose in its place, commonly referred to as *partitioned global address space* languages. The best-known examples are Co-Array Fortran [16], Unified Parallel C [8], and Titanium [22]. While their details vary greatly, these languages are similar due to their support for regular data distributions which are operated on in an SPMD style. They have the advantage of being easier to compile than HPF, but achieve this by shifting some of that burden back to programmers by requiring them to return to the fragmented programming model, writing per-processor code, and coordinating communication and synchronization explicitly (albeit using concepts that are significantly more abstract than MPI). These languages therefore offer an interesting midpoint between MPI and HPF in the tradeoff between programmability and performance.

OpenMP [6] is one of the few current parallel programming techniques that supports a non-fragmented, global view of programming. It is also comparably easy to apply to existing code since users can incrementally add annotations to code over time. OpenMP's primary disadvantage is that it assumes a uniform shared memory in its execution model and therefore typically cannot scale to large numbers of conventional processors. Due to these limitations, a hybrid MPI/OpenMP usage model has become common for clusters of SMPs in which MPI is used for the coarse partitioning between the nodes, and OpenMP is used to express the lighter-weight parallelization and synchronization on each node.

ZPL is another parallel language that supports a global view of parallel programming. It supports parallel compu-

tation via user-defined index sets called *regions* [2, 3]. Regions may be multidimensional, strided, and/or sparse, and are used both to declare distributed arrays and to operate on them in parallel. ZPL's region semantics are constrained so that all communication within a ZPL program is apparent in the syntax in the form of high-level array operators such as translations, reductions, and permutations.

3. The Cascade High Productivity Language Design

The history of programming languages has been a balance of abstraction to increase reuse and hence productivity with concreteness motivated by performance requirements. Chapel strives both to improve the performance of programs and to permit more abstraction to be used in the specification of those programs. This motivates the structure of this section, in which we first discuss *Concrete Chapel*, which allows an explicit high-level specification of locality and parallelism, while *Abstract Chapel* provides features for generic programming, supported by mechanisms for type and data structure inference, specialization, and prototyping tools.

3.1. Concrete Chapel

Concrete Chapel is a strongly typed object-oriented language with support for index *domains*, Fortran-like arrays, general data structures, data parallel operations, and explicit control of locality without resorting to a fragmented programming model.

The language is based on HPF's global view model which is very strong in dealing with common idioms for scientific computing at a high level of abstraction but avoids HPF's weaknesses of reliance on arrays as the only data structure and on "flat" parallelism. By adding general multithreaded programming and arbitrary data structures with object-level affinity Chapel greatly enhances its applicability to symbolic computing. Chapel's domains are generalizations of ZPL's regions, supporting nested parallelism, opaque index sets for building graphs, and a relaxed semantic model that emphasizes productivity over the identification of communication.

Chapel extends traditional sequential control constructs with the addition of explicitly parallel loops and a cobegin statement. The first of these specifies a space of iterations which may be executed concurrently. The second identifies a group of statements that are executed concurrently. These constructs may be nested and the amount of concurrency the programmer may express with them is unbounded.

The approach in Chapel differs in spirit from that of OpenMP. OpenMP creates heavyweight threads and has "work sharing" as a metaphor for managing parallelism. Chapel has no real notion of thread, just subcomputations that may be executed concurrently. By removing the concept of thread from the programming model, we eliminate a resource management concern. More importantly, thread management is no longer an aspect of the interface between code modules. Each module is free to express the concurrency natural to it.

Rather than binding work to threads, Chapel builds on a concept of *locale* to which both data and computation may have affinity. A locale corresponds to a portion of a computing system comprising both storage and processing. There is a presumption that co-locating a computation and the data it accesses in one locale will reduce latency and provide greater bandwidth to the data. However, unlike distributed memory models such as MPI, all data is accessible from any locale. Co-location is only a performance issue.

Chapel supports an "on" specification in HPF-2 style that directs execution of a subcomputation or allocation of a data object to a specific locale. This provides a low-level tool to manage locality by allowing a computation to be performed near the bulk of the data it accesses. The next two sections describe higher-level tools.

3.1.1. Domains A Chapel domain is a named index set that can be combined with distributions, linked to data structures, and provides a basis for the formulation of iterative processes. Domains are first-class objects that can occur as elements of data structures and can be passed to or returned by functions.

An important subclass are *Cartesian product domains* whose index sets are Cartesian products of integer intervals, such as $D = \{(i,j) \mid l_1 \leq i \leq u_1, l_2 \leq j \leq u_2\}$. Such domains support regular array data structures as in Fortran and form a basis for much of the data distribution machinery. Cartesian product domains support the ability to perform algebraic operations on indices, to express neighborhoods or subdomain boundaries in terms of simple index ranges and to reason about individual dimensions when distributing a multidimensional domain or specifying a nested iteration scheme. Cartesian product domains can be dynamically reshaped and redistributed; such operations affect all data structures linked to the domain at the given point of program execution.

Under appropriate constraints Cartesian product domains can deal with certain types of irregular applications (e.g., the multiblock problem discussed below) but in general a more flexible mechanism is needed for efficiently processing highly dynamic data structures such as graphs. For this purpose, Chapel introduces *opaque domains*, whose indices are system-generated objects, similar to pointers. For such domains, the Chapel runtime system provides an infrastructure for automatic partitioning and distribution.

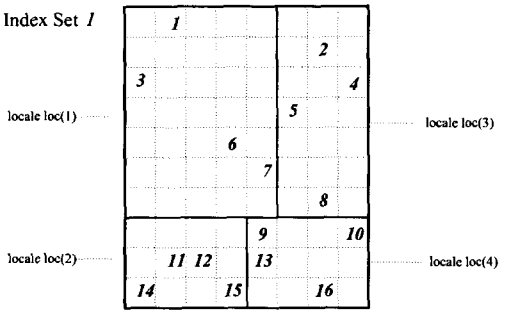

Figure 1. Sparse matrix distribution.

3.1.2. Data Distributions Programs in Concrete Chapel can organize their memory as a set of named virtual locales. A data distribution can then be introduced as a mapping from the index set of a domain to such a locale domain, where the set of all indices associated with one particular locale is called a *distribution segment*. Chapel defines new classes of intrinsic distributions, proposes user-defined extensions of the distribution mechanism, and provides special support for the distribution of opaque domains.

Chapel Distributions In addition to the data distributions of the HPF-2 Approved Extensions [11], which include general block and indirect, Chapel also supports user-defined distribution specifications. A new feature is the *general tiling distribution*, which generalizes the general block distribution by allowing the partitioning of a multi-dimensional index set into arbitrary rectangular subdomains, called tiles. A simple example for a general tiling distribution is the value-dependent distribution of a sparse domain, as illustrated in the example below.

Example: Sparse Matrix Distribution
Consider a Cartesian product domain for a sparse matrix with index set $I = (1:10, 1:8)$, where only non-zero elements are explicitly specified (Fig. 1). The locale domain is given by $loc(1:4)$. The figure illustrates a general tiling distribution constructed to balance the number of non-zeroes in each distribution segment. The resulting subdomains are $[1:7, 1:5]$, $[8:10, 1:4]$, $[1:7, 6:8]$, and $[8:10, 5:8]$. ◇

Example: Multiblock Code
Multiblock codes model geometrically complex objects such as aircraft by a set of interacting structured grids that are connected in an irregular manner. All grids can be processed independently in parallel, with updates of boundaries carried out periodically. The number of grids, their sizes and their interaction patterns are generally determined at runtime; different grids may have widely different sizes and shapes.

```
var MB : domain = [1..n_grids];
class Grid {
   var D : domain(1) distribute(block);
   var low, high : integer
   var data : array D of float;
   function do_distribute {
      distribute(D) on(Locales(low..high))
      allocate(data);
   }
   function solve;
}
var grids : array MB of Grid;
for i in MB {
   var s : domain(1) = read_shape();
   grids(i) = Grid(D=s);
}
setup(grids);
partition(grids);          -- determine low,high
forall g in grids
   g.do_distribute();

while (...) {              -- not terminated
   pre_process(grids);     -- update boundaries...
   forall g in grids on(Locales(g.low))
      g.solve()
}
function Grid.solve {
   forall i in D
      solve(i)...          -- implicitly on some locale
}
```

Figure 2. Parallel processing of a distributed multiblock grid collection.

Fig. 2 illustrates a simplified Chapel approach to a parallel solution of the multiblock problem. We distribute component grids to contiguous disjoint subsets of locales based on their size. This ensures that grids can be processed concurrently and that each individual solver can be processed efficiently in parallel. The multiblock grid collection is represented by a one-dimensional array, *grids*, with *n_grid* elements of type *Grid* for the individual component grids.

After determining the shape of all component grids, the grid collection is prepared for processing by a call to the function *setup*. This defines the boundary of each grid and initializes its data. The subsequently called *partition* routine determines the parameters, *lo* and *hi*, for the distribution of all component grids. The *while*-loop is inherently sequential. After pre-processing the grid collection, the *forall*-loop activates the solver for each component grid in parallel. The *on*- clause establishes affinity: each solver is executed on the locale subset of its associated grid. The individual solvers are parallel programs in their own right, which exploit the processing capabilities in their respective locale sets. ◇

```
class Tree {        -- k-ary trees
    var k : integer;
    var D : domain(opaque) distribute();
    var CD : domain(1) = 1..k;        -- child domain
    type Node;
    var nodes : array D of Node;
    var children : array D of array CD of index(D);
    class Node {
        var id : index(D);
        function add_child(i: index(CD), c: Node) {
            children(id,i) = c.id;
        }
        function child(i: index(CD)) : Node {
            return nodes(children(id,i))
        }
    }
    function newnode : Node {
        var n : index(D) = D.new();
        nodes(n) = Node(id=n);
        return nodes(n);
    }
}
var tree : Tree(k=num_children);
var root : Tree.Node = tree.newnode();

distribute(tree.D) on(Locales);

var data : array tree.D of datatype;
```

Figure 3. Using opaque domains to implement a k-ary tree.

Opaque Domains Opaque domains support the allocation and distribution of dynamic and irregular data structures. Their elements—opaque indices—are system generated objects, which identify instances of data associated with the domain and store information about the locale assignment and possibly the relative "weight" of the data element in the overall structure. This provides the system with sufficient information for automatically distributing (and possibly redistributing) such a domain.

Figure 3 shows a simple use of an opaque domain to manage the nodes of a k-ary tree. The multiblock example discussed above could also be formulated using opaque domains, based on a dynamic allocation of individual grids and automatic support for partitioning (rather than the explicit control expressed by the function *partition*).

3.2. Abstract Chapel

The definition of "abstract" that applies here is *disassociated from any specific instance*. In particular we want to enable and encourage programs to be written with explicit algorithms but with abstract data structures.

```
function search(v) {
    dfnumber(v) = count;
    lowlink(v) = count;
    count += 1;
    stack.push(v);
    on_stack(v) = true;
    for w in neighbors(v)
        if (not is_old(w)) {
            search(w);
            lowlink(v) = min(lowlink(v), lowlink(w));
        } else if (dfnumber(w) < dfnumber(v) and on_stack(v))
            lowlink(v) = min(lowlink(v), dfnumber(w));
        }
    if (lowlink(v) == dfnumber(v)) {
        var r = new_component();
        var x;
        do {
            x = stack.pop();
            on_stack(x) = false;
            add_node(r,x);
        } while (dfnumber(x) > dfnumber(v));
    }
}
```

Figure 4. A search function for finding strongly connected components.

Chapel will focus on providing support for abstracting three fundamental aspects of data structures: component types, iteration, and mapping from sets to variables. For example, consider Tarjan's algorithm for finding strongly connected components shown in Figure 4. The context for this algorithm is that the value of v identifies a node in some graph and *neighbors* defines an edge relation for that graph. There are a number of *maps* from the graph nodes to various variables that hold state for the algorithm. These are *dfnumber*, *lowlink*, *is_old*, and *on_stack*. Within the language, the *array...of* type constructor is used to identify maps. There is also a loop that iterates over a set of nodes identified as *neighbors* of v. In addition, two unbound functions, *new_component* and *add_node*, abstract the actions we should take when a new strongly connected component is identified and a new element is added to a component. Finally, the algorithm uses the unbound objects *count* and *stack* which carry state between activations of the function.

The above algorithm is quite abstract. To make it concrete we would need to provide implementations for the various maps, the iterator, and bindings for the unbound operations. The object-oriented paradigm provides a framework for some of these issues. In particular, we can encapsulate this function in a class that binds free symbols. However, existing strongly-typed object-oriented languages give poor support for adapting the mapping and iteration functions

to client contexts. Furthermore, while here it is enough to specify that the set of nodes forms a graph and has a neighbor relation, any further definition of the graph would limit the applicability of this function and curtail its potential for reuse. Chapel extends the common object-oriented features in three ways to encourage abstraction:

1) *Abstraction of types* Similar to the concept of templates in C++ and generic packages in Ada, we permit types to be omitted from program fragments. Types will be inferred from usage and different contexts may require separate instantiations of various functions and classes each tailored to callsite specifics. In the example above, variables *v*, *w*, and *r* might be simple integers, pointers to objects, or elements in some domain. Each is a valid choice depending on context. We permit type variables to be declared which can be used to express constraints amongst complex types [7].

2) *Abstraction of iteration* Iteration over sets of objects and values is another fundamental operation but most languages provide little in the way of abstraction for this operation. We borrow from CLU [12] the concept of an *iterator* which returns a sequence of values or object references. These iterators are named, have parameters and can be bound to objects like other functions. The primary difference is that they return a sequence of values rather than a single value. In this example, the iterator *neighbors* insulates this code from the details of the graph abstraction. Iterators are integrated with the concurrency mechanisms as well.

3) *Abstraction of maps* We use the term *map* to refer to a function from some domain to a collection of variables. It generalizes the concept of "array" which is based on Cartesian product domains. The domain of a map may be an integer tuple, an opaque domain, the values of some type (implemented with a hash table), or the values returned by some iterator. A goal of Chapel is to provide default implementations for various sorts of maps to facilitate rapid prototyping yet also to provide an interface that allows construction of context-efficient implementations without disturbing the algorithm.

Another productivity goal of Chapel is to expand from type inference to *data structure inference*. This means we would like to have the programmer identify a general category for an object and have the system select a reasonable implementation based on use. For example, the user identifies *dfnumber* as a map, simply as an array indexed by a type, and the system selects a suitable implementation depending on its knowledge of the domain. Similarly, we have a category *seq* that consists of objects that have iterators. There will be a library of *seq* implementations, generic with respect to element type, but with different sets of methods such as *push* and *pop*. The system will automatically select an instantiation based on usage but the programmer is free to override this decision or to directly implement the abstractions based on application specific knowledge that admits a more efficient solution.

```
class StrongComponents {
   type nodetype;
   iterator neighbors(node :nodetype) :nodetype {
      return node.neighbors;
   }
   type nodeset : seq of nodetype;
   var components : seq of nodeset;
   function new_component : nodeset {
      var x = nodeset();        -- create new nodeset
      components.insert(x);
      return x;
   }
   function add_node(r :nodeset, n :nodetype) {
      r.insert(n);
   }
   var dfnumber, lowlink : array nodetype of integer;
   var on_stack : array nodetype of boolean;
   function is_old(n :nodetype) : boolean {
      return lowlink(n) < 0;
   }
   var count : integer;
   var stack : seq of nodetype;
   function find_components(nodes) : seq of nodeset {
      forall n in nodes {
         on_stack(n) = false;
         lowlink(n) = -1;
      }
      stack.init();
      count = 0;
      for n in nodes
         if (not is_old(n))
            search(n);
      return components;
   }
   function search(n :nodetype) { ... }
}
```

Figure 5. An abstract class for finding strongly connected components in a directed graph.

We illustrate these concepts in a possible encapsulation of the *search* function shown in Figure 5. This class is generic with respect to the type parameter *nodetype* which must be specified when an instance of this class is instantiated. The type of the formal parameter *nodes* to method *find_components* is also left generic. It could be an array or other domain or an object with a default iterator. The local type variable *nodeset* constrains the return type of *new_component* to match the first parameter of *add_node*.

Default implementations for the various sets and arrays bound to this class will be provided. The default implementation for *stack* includes the *push* and *pop* methods. A client

```
class MyStrongComponents with StrongComponents {
   where class nodetype { var dfnumber, lowlink : integer;
                          var on_stack : boolean;
                          var stack_next : nodetype; }
   function dfnumber(n : nodetype) { return n.dfnumber; }
   function lowlink(n : nodetype) { return n.lowlink; }
   function on_stack(n : nodetype) { return n.on_stack; }
   var stack : class {
      var top : node;
      function push(n : nodetype) {
         n.stack_next = top;
         top = n;
      }
      function pop :nodetype {
         var x = top;
         top = x.stack_next;
         return x;
      }
      function init { top = nil; }
   }
}
```

Figure 6. A specialization where auxiliary fields on graph nodes are used instead of side data structures.

is free to specialize this class definition by providing alternate implementations of any or all of these definitions. The category of a symbol is allowed to change as well so that array *dfnumber* could be altered to be a function returning a variable.

An example of such a client is shown in Figure 6 where the system implementations of various arrays are replaced with direct access to variables on graph nodes. We add a type constraint on *nodetype* that identifies these fields and provide a new definition of *stack*.

The primary implementation challenge is to manage the cost of recompilation and optimize the selection of implementations to avoid unnecessary overheads. Unlike other high-level languages, however, runtime performance is not completely at the mercy of a smart compiler: where optimization fails, the language allows programmers to provide concrete, hand-crafted implementations of data structures that will be competitive with today's solutions. These implementations can be provided without disturbing the basic algorithms. In particular, they could be target-dependent, providing a level of portability for the bulk of an application not currently available.

While the research focus of the concrete language is to explore locality-aware multithreading as a programming model, for the generic aspects we are interested in achieving much of the ease of use and expressiveness of dynamically typed languages but with the performance advantages of compile-time type checking. We are also hoping to bridge high-level languages like SETL [19] and more production-oriented languages by specializing reusable algorithms built on abstract data structures. A synergy between these goals is provided by the common notion of a domain as a mechanism both to define maps and iteration and to distribute data for locality.

3.3. Implementation Strategy

The initial implementation of Chapel will take a source-to-source compilation approach in order to provide a portable implementation that avoids as many architectural assumptions as possible. This implementation will most likely generate C code with calls to a portable communication interface such as MPI or ARMCI [15], in order to maximize the number of parallel architectures on which it can be run. This compiler will be made open source early in its development cycle in order to encourage evaluation of the language, experimentation with its features, and rapid feedback from potential user communities. As the implementation matures, we hope to engage the broader community to help with its optimization, upkeep, and evolution.

One of the primary challenges with this initial implementation will be to achieve performance levels that make Chapel attractive not only for its productivity features, but also for performance-oriented runs. This will be a stiff challenge given the lack of support for latency tolerance and multithreading on current architectures, but our hope is that Chapel's language-level support for abstractions such as domains and locale views will give the implementation the opportunity to cache pieces of performance-crucial information with the runtime objects representing these concepts.

Meanwhile, as the Cascade architecture moves toward production, a second implementation effort will take place to generate code that takes maximal advantage of its unique features. This implementation will target either the machine's assembly language, or the front-end of Cascade's conventional C and Fortran compilers.

Chapel's abstract features will be implemented via a global type inference engine that strives to determine variable types and values within a user's program. The compiler will then specialize routines to generate optimized code based on the results of that analysis. This implementation effort is already underway and is based on compilation techniques developed in the context of the Concert Compiler [17].

4. Architecture Implications

Key concepts of high-productivity programming on which the design of Chapel is founded include multi-

threaded fine-grain parallelism and locality-aware programming. These are areas where the lack of adequate architectural support in conventional architectures has led to severe performance problems. Here we will discuss how the Cascade architecture addresses these issues.

1) Implementation of fine-grain parallelism with an unbounded number of threads implies a runtime system capable of virtualizing processors. This requires multiplexing threads onto available processors, hardware support for synchronization, and mechanisms to reduce the overheads of thread scheduling.

Modern processor architecture has focused almost exclusively on the performance of single threads of control. Large register sets, data caches, memory disambiguation and branch prediction are all focused on a single thread, resulting in a large execution state and expensive context switches. Such processors are most effective when executing programs with a high ratio of number of operations to memory bandwidth requirements. Cascade provides such "heavyweight" processors but also includes a second class of "lightweight" processors.

Lightweight processors are optimized for programs rich in short threads and synchronization, or which are data intensive. These processors are implemented in the memory subsystem and while they are not very powerful individually, there are many of them. Lightweight processors use a form of hardware multithreading to tolerate latency for remote accesses and synchronization.

2) Chapel supports the distribution of data structures to exploit spatial locality. At the same time, high bandwidth is needed for data structures for which there is no natural distribution. This is particularly important for expanding parallel programming to areas beyond simulation of physical systems where decompositions are often implied by system structure.

The Cascade architecture is a shared address space system that supports multiple views of physical memory. In one view, which supports non-uniform spatial locality, the system consists of an arrangement of locales, each of which contains a heavyweight processor and a collection of lightweight processors in the memory. Neighborhoods in the virtual address space are preserved by the hardware mapping to physical addresses.

To support data structures that are shared but have no natural algorithm-induced distribution, Cascade supports a uniform-access view of memory. In this view, small consecutive blocks of memory are distributed by applying a hashing function to the virtual addresses. This technique was used in the IBM RP3 and the Cray MTA to ensure high bandwidth to data aggregates from a set of processors regardless of access pattern. For example, a 3-D data set can be partitioned into parallel 1-D "pencils" and be accessed concurrently in any dimension with no loss of bandwidth.

These two models allow spatial locality to either be ignored or exploited. It can be ignored at small scales that have adequate bandwidth or that have no viable decomposition. It can be exploited at large scale where a reasonable distribution for the data can be achieved.

3) The final aspect of Chapel is the the need for remote thread creation or, equivalently, message-driven computation to allow threads to execute in the locale of the data they are accessing. This concept has many antecedents including Actor programming [1], the Chare Kernel [20], and Active Messages [21].

All processors in the Cascade system are able to execute a *spawn* primitive that will create a new thread running in a lightweight processor anywhere in the system. This mechanism directly supports the programming idiom of creating a new thread running on a particular locale to exploit spatial locality or reduce the latency of transactions against remote data.

This basic model is then extended with a software layer so that suitable threads can be executed by an adjacent heavyweight processor to exploit temporal locality. Heavyweight processors service a local work-queue of threads ready to run. Prior to insertion on this queue, remote data can be prefetched and synchronization resolved to minimize the impact of these issues on heavyweight processors that are not optimized for them.

These are areas where the Cascade architecture will provide non-commodity support for the Chapel programming model. They are not the only issues needed for a successful Petaflops-scale system nor are they the only ways to exploit Cascade's architecture but these are key areas where current systems are insufficient.

5. Summary

This paper described the design of Chapel, the Cascade High Productivity Language. Chapel pushes the state-of-the-art in languages for high-end computing by combining the goal of highest possible object code performance with a high-level user interface for programmability. While the concrete language allows explicit control of parallelism and locality without the need to resort to a fragmented memory model, the abstract language frees the user from many details of type and data structure specification. Chapel can be implemented on any parallel system, however the full range of the language will require architectural support for achieving its performance goals beyond that offered in conventional machines. Such support can be realistically expected from many future Petaflops-scale architectures, and it will be provided specifically by the Cascade architecture

in the areas of fine-grain multithreading, locality-aware programming, and message-driven computation.

Higher-level programming models like that provided by Chapel are one of several innovations that may be needed to more effectively program highly parallel systems. Others include new approaches to compiler, runtime system, and tool design, as well as enhanced automatic support for algorithm development, fault tolerance, and correctness and performance debugging. The conventional separation between compiler, runtime system, and tools is breaking down as a result of emerging concepts such as feedback-oriented and just-in-time compilation, dynamic instrumentation, and AI-based approaches to hotspot detection and performance tuning. Autonomous agents may monitor the execution of a parallel program at runtime, providing feedback about memory leaks, excessive thread generation, ineffective access patterns to distributed data structures, or potential parallel hazards. Likewise, a software infrastructure may transparently deal with such situations, triggering recompilation of critical program loops, redistributing data structures, or preventing deadlocks by imposing constraints on resource accesses.

References

[1] G. Agha, W. Kim, and R. Panwar. Actor languages for specifi cation of parallel computations, 1994.

[2] B. L. Chamberlain, E. C. Lewis, C. Lin, and L. Snyder. Regions: An abstraction for expressing array computation. In *ACM/SIGAPL International Conference on Array Programming Languages*, pages 41–49, August 1999.

[3] B. L. Chamberlain and L. Snyder. Array language support for parallel sparse computation. In *Proceedings of the 2001 International Conference on Supercomputing*, pages 133–145. ACM SIGARCH, June 2001.

[4] B. Chapman, H. Zima, and P. Mehrotra. Extending HPF for advanced data-parallel applications. *IEEE parallel and distributed technology: systems and applications*, 2(3):59–70, Fall 1994.

[5] B. M. Chapman, P. Mehrotra, and H. P. Zima. Programming in Vienna Fortran. *Scientifi c Programming*, 1(1):31–50, Fall 1992.

[6] L. Dagum and R. Menon. OpenMP: an industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January–March 1998.

[7] M. Day, R. Gruber, B. Liskov, and A. C. Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 156–168, 1995.

[8] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. *UPC Language Specifi cation*, October 2003.

[9] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. W. Tseng, and M. Y. Wu. Fortran D language specifi cation. Technical Report CRPC-TR90079, Houston, TX, December 1990.

[10] High Performance Fortran Forum. High Performance Fortran language specifi cation. *Scientifi c Programming*, 2(1–2):1–170, Spring–Summer 1993.

[11] High Performance FORTRAN Forum. High performance fortran language specifi cation, version 2.0. Technical report, Rice University, 1997.

[12] B. Liskov, A. Snyder, R. R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8), August 1977.

[13] P. Mehrotra and J. V. Rosendale. Programming distributed memory architectures using Kali. In *Advances in Languages and Compilers for Parallel Computing*. MIT Press, 1991.

[14] P. Mehrotra, J. V. Rosendale, and H. Zima. High Performance Fortran: History, status and future. *Parallel Computing*, 24(3):325–354, May 1998.

[15] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586:533–546, 1999.

[16] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.

[17] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1994.

[18] H. Sakagami, H. Murai, and M. Yokokawa. 14.9 TFLOPS three-dimensional fluid simulation for fusion science with HPF on the earth simulator. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14. IEEE Computer Society Press, 2002.

[19] J. T. Schwartz. Automatic data structure choice in a language of very high level. *Communications of the ACM*, 18(12):722–728, December 1975.

[20] W. Shu and L. V. Kale. Chare kernel - A runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11(3):198–211, 1991.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[22] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfi nger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

[23] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.

[24] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - A language specifi cation, version 1.1. Technical Report ACPC/TR 92-4, Dept. of Software Technology and Parallel Systems, University of Vienna, Mar. 1992.