

ORNL/TM-2007/122

Expressing POP with a Global View Using Chapel: Toward a More Productive Ocean Model

July 2007

Richard F. Barrett, Stephen W. Poole, and Sadaf R. Alam

DOCUMENT AVAILABILITY

Reports produced after January 1, 1996, are generally available free via the U.S. Department of Energy (DOE) Information Bridge:

Web Site: <http://www.osti.gov/bridge>

Reports produced before January 1, 1996, may be purchased by members of the public from the following source:

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone: 703-605-6000 (1-800-553-6847)
TDD: 703-487-4639
Fax: 703-605-6900
E-mail: info@ntis.fedworld.gov
Web site: <http://www.ntis.gov/support/ordernowabout.htm>

Reports are available to DOE employees, DOE contractors, Energy Technology Data Exchange (ETDE), and International Nuclear Information System (INIS) representatives from the following sources:

Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831
Telephone: 865-576-8401
Fax: 865-576-5728
E-mail: reports@adonis.osti.gov
Web site: <http://www.osti.gov/contact.html>

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL-2007/122

**EXPRESSNG POP WITH A GLOBAL VIEW USING
CHAPEL: TOWARD A MORE PRODUCTIVE OCEAN
MODEL**

Richard F. Barrett, Stephen W. Poole, and Sadaf R. Alam

Date Published: July 2007

Prepared by
OAK RIDGE NATIONAL LABORATORY
P. O. Box 2008
Oak Ridge, Tennessee 37831-6285
managed by
UT-Battelle, LLC
for the
U. S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Contents

List of Figures	v
Abstract	vii
1 Introduction	1
1.1 Productivity Discussion	1
1.2 Research Contributions	2
1.3 Report outline	3
2 Overview of Chapel	3
2.1 Syntax and Semantics	5
3 Overview of POP	6
4 Computing surface pressure: the 9-point stencil	8
4.1 Chapel implementation	10
4.2 Productivity analysis	11
5 The 5-point stencil	13
5.1 Chapel implementation.	13
5.2 Productivity analysis	14
6 A Brief Note Regarding Performance Expectations	15
7 Conclusions and Future Work	16
A POP's MPI halo exchange	19
B POP's Co-Array Fortran halo exchange	20

List of Figures

1	Global view vs. fragmented view parallel programming model. .	4
2	Chapel reduction operator	7
3	POP's matrix-vector product posed as 2d 9-point stencil computation.	9
4	POP's matrix-vector product in Chapel using parameterized tuple arithmetic.	11
5	The matrix coefficients associated with each grid point are stored as 3×3 blocks in the "array of arrays" defined domain <code>Coeff</code>	12
6	POP's matrix-vector product in Chapel using the reduction operator.	12
7	The POP 2d 5-point stencil.	13
8	Chapel 2d 5-point stencil.	14
9	Runtime conditional stencil configuration.	15

Abstract

Chapel is one of the programming languages being developed as part of the DARPA High Productivity Computing Systems (HPCS) program. In this paper, we describe how this language could be used to express some important mathematical structures and algorithms found in the Parallel Ocean Program (POP), a widely used ocean circulation model. We compare the current Fortran+MPI and Co-Array Fortran implementations with those using the higher level global view provided by Chapel. Our analysis suggests that the Chapel implementations can capture the application developer's view of an algorithm and offer algorithmic polymorphism, resulting in a significant improvement in code development productivity for the ocean modeler and other scientists whose work involves similar computations.

1 Introduction

The objective of the DARPA High Productivity Computing Systems (HPCS) program[11] is to provide computational scientists with a more effective high performance computing environment. Productivity, then, must encompass many ideas, including programmability, performance, portability, and robustness. As part of this effort, vendors are developing new programming languages that are intended to provide scientific application developers a means for easily expressing their algorithms in a form that also enables efficient computation on parallel distributed memory computing systems. In this paper we report on our investigation into how one such language, Chapel[4], addresses these issues within the context of a widely used large scale scientific application program.

In this study, we identify and discuss Chapel constructs that can help the computational scientist express algorithms in a manner that keeps the focus on the algorithms and applications rather than on the details associated with the parallel computing architecture. In particular, we target key mathematical algorithms used by a production-level DOE Office of Science application, Parallel Ocean Program (POP)[23], which models ocean circulation. The selected computations are involved in the solution of partial differential equations defined by the model, and thus this work will be relevant to a broad range of scientific areas. In particular, we compare the current Fortran with MPI[26, 16] and Co-Array Fortran (CAF)[22] implementations of a key component of the computation of surface pressure and a fundamental computation used throughout the code with those we can create using the higher level global view of computation provided by Chapel.

1.1 Productivity Discussion

As stated above, productivity encompasses many ideas. In code development, this must be centered around user requirements and goals as well as the context of individual use. We can discuss, and even measure, ideas for comparisons from the perspective of the code developer.

Programmability is intended to capture some notion of the level of effort required to create, maintain, and extend a program. One measure is the number of lines of source code (SLOC), which includes executable statements, declarations, and comments*. We measure this using CodeCountTM[24], which among other things, distinguishes between physical and logical lines

*Comments are relevant since, as we'll see, the less obvious the block of code, the more commenting is required.

(the latter ignores line breaks within a language statement), whole and embedded comments, and blank lines.

We qualify this metric with the notion of expressibility. We define expressibility as the ability of the code developer to articulate the necessary computations in a manner that is natural to the particular task *and* conveys some notion of intent that may be exploited by the compiler. Furthermore, algorithmic polymorphism offered by Chapel enables code developers to express variants of an algorithm in a unified manner. In other words, we are attempting to quantify the number of effective lines of code as a productivity metric. Although still somewhat subjective, this gives us a basis to discuss the quality of the code. Nevertheless, the final measure of programmability must be decided by the code developer.

A language that provides expressive syntax and semantics should result in programs with fewer coding errors, is easier to debug, is easier to modify and extend, requires fewer comments, and achieves strong performance on a breadth of computing architectures.

1.2 Research Contributions

Our goal in this work is to investigate the ability of the code developer to express an algorithm, contrasted with the requirements for dealing with the intrusion of the details associated with the parallel processing environment. Although we do this within the context of a particular application (POP), we examine computations that are found in a broad range of scientific applications.

At the same time it is important to note that our goal is not to provide a tutorial for creating a Chapel implementation of POP nor even to suggest that this need be undertaken. Instead, POP provides a real-world example that is representative of the coding effort and style required by the message passing model using the Fortran programming language, thus serving as a basis for comparison with Chapel constructs.

Where possible, we verify the correctness of our Chapel implementations using a pre-release version of a compiler that provides rudimentary functionality compared to the language specification. We also use functionality that is still under development, and therefore cannot be verified by the compiler. In this case, great care has been taken to ensure correctness of this code, including verification by the Chapel development team[5]. In doing so, our experiences feed back to the Chapel team in a manner that helps them evolve the language in directions that are useful to scientific applications and away from those that are not.

1.3 Report outline

We begin by presenting an overview of Chapel, with a focus on the syntax and semantics used for this work along with an introduction to alternate programming languages. After a brief overview of POP, we examine important computations used in the model, comparing and contrasting implementations written using the message passing model, Co-Array Fortran, and Chapel. We speculate on performance issues, then offer our conclusions and discuss our future research.

2 Overview of Chapel

Scientific applications designed for use on high performance computing architectures are most often implemented based on functionality found in the Message Passing Interface (MPI)[26, 16] specification. This approach requires the code developer to explicitly manage the distribution and movement of data among the parallel processes, each of which has its own distinct address space.

To address some of the difficulties perceived with this approach, alternative programming languages based on the Partitioned Global Address Space (PGAS) model have been proposed. For instance, Co-Array Fortran (CAF)[22] makes data globally accessible via co-array load and store semantics, though it still places the responsibility for distributing and moving the data between the parallel processes on the code developer. Unified Parallel C (UPC) [27] extends the C programming language to include a shared address space view of computation, but it inherits C’s limited support for arrays, with no real support for multidimensional arrays. Titanium[19] extends JavaTM to include an explicit parallel Single-Program-Multiple-Data (SPMD) model.

A common feature of these programming models is that they provide a “fragmented view” of parallel computation in that they require the developer to explicitly manage the interaction of the parallel processes as well as the overall data layout. In contrast, “global view” models abstract the parallel processes from the view of the developer in an effort to simplify the expression of a given algorithm as a parallel computation[†]. OpenMP[10] presents such a view through the definition of a set of compiler directives

[†]Chapel also provides access to locality, lower level constructs, and a task parallelism capability. Although these capabilities are useful and can supplement the global view, they are not needed for our current purposes.

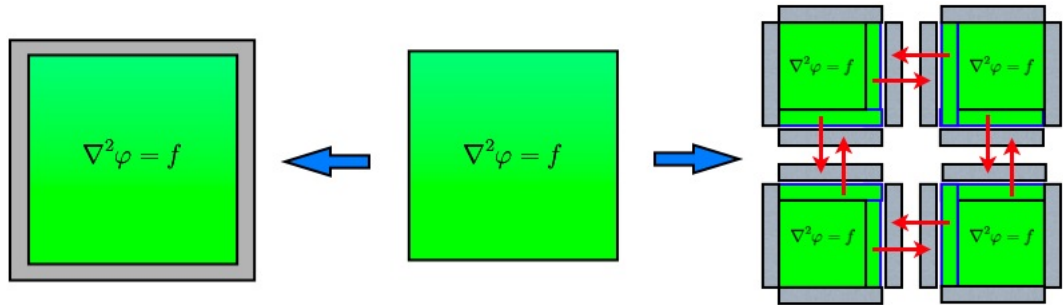


Figure 1: Global view vs. fragmented view parallel programming model. Consider a partial differential equation (here Poisson’s Equation) defined on a two dimensional domain, illustrated by the center picture. The fragmented view configuration for applying a solution algorithm on a parallel processing computer is shown on the right. Here the code developer must manage the interaction of the parallel processes as well as the overall data layout, including explicit control over the sharing of data among the individual blocks. This is usually accomplished by surrounding each block with a “halo” in order to control data movement (as indicated by the arrows) and maintain coherency. A global view language such as Chapel captures data associated with the problem in a single structure which it (as well as a fragmented model) may then surround with space for the physical boundary conditions. Although the language may provide a means for conveying information regarding parallelism in the problem (Chapel does[12]), the code developer is not responsible for distributing and sharing data amongst the parallel processes.

and associated syntax, but is limited to regions of physically shared memory in a node. It may be combined with MPI to link multiple distributed shared memory regions such as on a cluster of SMP nodes, which then introduces the fragmented view. High Performance Fortran (HPF)[18] provides a global view, but is constrained to the SPMD model and supports only a single data structure (the Fortran array). This rules out a broad range of problems defined on irregular domains as well as the implementations described in Section 5.

One goal of the HPCS language effort is to combine the strengths of these existing programming models while avoiding their weaknesses. Chapel pursues this goal by providing a global view of parallel programming based around the definition of a *domain*. The domain is a construct that provides

the code developer with a means for configuring data structures that enable a more natural mapping of computations to the parallel processes, including distribution of data and associated inter-process data sharing. The overall goal is to combine a global view of the program with the tools necessary for injecting high-level programmer “intent” that the compiler cannot easily discover in more traditional programming models.

At the time of this writing, the Chapel language specification[9] is at version 0.750. A prototype compiler (pre-release version 0.4) has been provided to a small group of programmers who are gaining experience and providing feedback to the Chapel developers.

Two other notable global view language development efforts are also underway. Fortress[1] endeavors to present a mathematically based syntax to the code developer. X10[6] extends JavaTM. We are investigating these languages in a manner similar to that described in this report.

2.1 Syntax and Semantics

Unlike languages such as HPF, CAF, UPC, and Titanium, Chapel does not extend an existing serial language to include a parallel processing capability. Instead, developers began with a clean slate so that a global view of a parallelism may be constructed free of existing constraints.

Chapel syntax and semantics are sufficiently similar to Fortran, C, and C++ so that a scientific application developer familiar with these languages can quickly become comfortable with this new language. Like C, executable statements are terminated by a semi-colon, braces define executable blocks, variables can be declared (and initialized) within executable blocks, and variables can be re-cast. Like Fortran, multidimensional array indices, bounded by user defined values, are indexed within parenthesis (but like C are declared using brackets), and modules define name spaces and are included in the compilation unit via the `use` statement. Like C#, C++, and Java, object-oriented programming is possible with Chapel, and comments can either be captured within a block by slash-star star-slash or prepended by double slash. Chapel also bears some similarities to HPF, ZPL[3], and the Cray Multi-threaded Architecture (MTA) extensions to C and Fortran[8].

Chapel provides a global view of parallelism through the definition of a *domain*, which describes data using an index set, and optionally a distribution, associated arrays, and functions (called iterators). A domain may induce another domain, as a subset, superset, or a translation. This can be useful in defining a space that includes the physical domain as well as space for applying boundary conditions. These constructs enable a parallel

perspective of the computation regardless of the underlying mechanisms for parallel execution.

Arithmetic domains are rectilinear sets of Cartesian indices of an arbitrary rank. Sparse domains are subsets of some base domain, supporting graph-based algorithms, sparse matrix computations, and as we will show, certain kinds of difference stencils. The `forall` iterator is the construct for expressing a parallel computation over a domain's indices. *Tuples* aid readability by combining per-dimension index counters (`i` and `j`) into a single variable, say `k`, which can then be operated on using arithmetic.

Note that although the focus of the work reported herein is based on a data parallel model, task parallelism is also supported and can coexist within the same program.

Reduction operator

Scientific applications often require the aggregation of values. For example, in stiff systems, a Courant condition determines the length of the next time step; in solving linear systems using Krylov subspace methods, inner products and vector norms must be computed. When the data is distributed across parallel processes, these values are typically computed independently by each process, followed by a summation across all processes in order to compute the global value. This reduction operation is explicitly supported by protocols such as MPI and OpenMP. Like MPI, Chapel provides several reduction operations as well as a means by which the code developer may define an operator[13]. Whereas MPI provides a mechanism for applying the reduction using a subset of the parallel processes (the communicator), Chapel defines reductions on subsets of domains, which, as we will see in Section 4.1, provides a more naturally expressive semantic, both from a code as well as compiler perspective. Figure 2 illustrates the use of the reduction operator, computing the maximum difference between grid values, with the view restricted to those points in the physical space, i.e. excluding the boundaries.

3 Overview of POP

POP (Parallel Ocean Program) is a freely available ocean circulation model[23] developed at Los Alamos National Laboratory, currently at production version 2.0.1. It also serves as the ocean component of the Community Climate System Model (CCSM[2]). POP models ocean circulation by solving time dependent equations describing fluid motion in three dimensions. The

```

1  const
2    PhysicalSpace = [1..m, 1..n, 1..p], // Grid points in the 3d physical domain.
3    AllSpace = PhysicalSpace.expand(1); // Physical domain plus boundary.

4  var
5    x, y : [AllSpace] real;

6  // Note: x and y are allocated in AllSpace, operated on in PhysicalSpace.

7  delta = max reduce abs ( x[PhysicalSpace] - y[PhysicalSpace] );

```

Figure 2: Chapel reduction operator

barotropic phase computes surface pressure in two dimensions. The baroclinic phase resolves interactions between levels in the vertical direction. These two phases define most of the work in each model time step.

Computation takes place on an orthogonal curvilinear coordinate system on a dipole grid[25]. In order to improve resolution of the Arctic ocean region, a tripole grid[21] option was added to version 2, leading the developers to reconfigure the code in a manner that makes the computations of interest here less amenable to clear comparison. Because our goal is aimed at understanding how Chapel might be used in the context of common operations in scientific applications rather than in providing a guide for creating a Chapel implementation of POP, we base our work on the bipole-only previous version 1.4.3. This version continues to be used for performance evaluations[20, 15], and will serve as a basis for comparison in the future. Additionally, although not officially supported, a Co-Array Fortran implementation of the 9-point stencil (examined in section 4) has been prototyped[‡] for version 1, which provides another basis for comparison.

The Fortran-MPI implementation is written using 36,702 total lines of Fortran, of which 22,447 are classified as physical SLOC and 16,770 are classified as logical SLOC. It is well commented, with a percentage of comments (whole and embedded) to physical and logical SLOC of 60% and 101.5%, respectively. The actual message passing functionality is contained in five files[§], listed in table 1. This accounts for 15% of the total number of lines of code in POP, with 14% and 12.7% classified as physical and logical SLOC.

The table does not completely capture the coding requirements for using MPI in POP. For example, source code not listed in the table has been

[‡]Written by Pat Worley of ORNL and John Levesque of Cray Computer Corp.

[§]The sole exception is the file that enables coupling of the ocean model to the other CCSM components, which we exclude from our SLOC metrics since it is not used in the stand-alone version of POP.

<i>Module</i>	<i>Total Lines</i>	<i>Cmnts</i>	<i>Data Decl.</i>	<i>Exec. Instr.</i>	<i>SLOC</i>		<i>Description</i>
					<i>Physical</i>	<i>Logical</i>	
<code>boundary</code>	986	358	52	327	590	386	Ghost cell update.
<code>communicate</code>	254	111	19	98	174	119	Process management
<code>global_reductions</code>	1195	641	219	237	721	489	Various reductions.
<code>io</code>	955	611	96	241	439	337	MPI collects, Fortran
<code>stencils</code>	2138	1134	184	593	1221	796	Stencil computation
<i>Totals</i>	5528	2855	570	1496	3145	2127	

Table 1: Lines of code classification of POP’s MPI functionality.

configured for such things as conditionals that limit execution to the master task and calls to global communication, such as reductions and broadcasts. This code accounts for approximately 700 logical SLOC. On the other hand, some of the SLOC in the routines listed in Table 1 would be retained, such as the stencil computations. (We will see, though, that this will be minimized using Chapel’s polymorphic capabilities.) Barring a first-principles rewrite of POP in Chapel necessary for a definitive comparison, we claim that somewhat more than 10% of the code in POP’s MPI implementation is attributable to inter-process communication requirements induced by the fragmented view mode. Besides, as previously discussed, precision is not required as the SLOC metric doesn’t fully capture the relevant issues in the productivity of an implementation.

Finally we note that POP makes strong use of MPI functionality designed to clarify some basic tasks. For example, a two dimensional Cartesian logical processor grid is configured, which simplifies identification of nearest neighbors. Derived types are defined, a convenience in exchanging ghost cells (also known as halos). Still, the actual exchange of the boundaries requires significant coding (contained in the `boundary` module, listed above in Table 1).

4 Computing surface pressure: the 9-point stencil

Computation of surface pressure requires the solution of the 2-dimensional barotropic equations. The implicit solution method employed by POP configures an elliptic equation of the form

$$AF = B \tag{1}$$

where F is a field (in this case surface pressure) and A is the operator defined as

$$AF = a\nabla \cdot (H\nabla F) \quad (2)$$

where a is the cell area. For each model time step, a linear system of equations is generated of the form

$$Ax = b \quad (3)$$

for coefficient matrix $A \in \mathbf{R}^{n \times n}$ and vectors x and $b \in \mathbf{R}^n$.

The solution of this symmetric positive definite sparse system is computed iteratively using one of three options: preconditioned conjugate gradients (PCG)[17], conjugate residuals, or a diagonally preconditioned Richardson-Jacobi solver. PCG is the default algorithm, typically requiring around 200 iterations for convergence.

Each linear equation in the system is defined as a function of a grid point and the eight grid points immediately adjacent to it. Thus the matrix-vector product, computed as part of each iteration, may be formulated as a nine-point stencil sweep across the grid. The Fortran implementation of this computation is shown in Figure 3. Each processor owns arrays of size

```

1      real (kind=dbl_kind), dimension(imt,jmt), intent(in) ::
2      &   X, XOUT,           ! array to be shifted
3      &   CC,CN,CS,CE,CW,   ! weights in each of the nine directions
4      &   CNE,CSE,CNW,CSW

5      do j=jphys_b,jphys_e
6          do i=iphys_b,iphys_e
7              XOUT(i,j) = CNE(i,j)*X(i+1,j+1) + CNW(i,j)*X(i-1,j+1) +
8              &           CSE(i,j)*X(i+1,j-1) + CSW(i,j)*X(i-1,j-1) +
9              &           CN (i,j)*X(i ,j+1) + CS (i,j)*X(i ,j-1) +
10             &           CE (i,j)*X(i+1,j ) + CW (i,j)*X(i-1,j ) +
11             &           CC (i,j)*X(i,j)
12          end do
13      end do

```

Figure 3: POP’s matrix-vector product posed as 2d 9-point stencil computation.

$\text{imt} \times \text{jmt}$ for storing the matrix coefficients and grid points as well as for ghost cells (also called halos) filled using inter-process communication. The subdomain of grid points actually owned by a parallel process is a subset of that space, running from `iphys_b` to `iphys_e` to `jphys_b` to `jphys_e`. The coefficients for each (i, j) linear equation in the system are stored in separate arrays, e.g. $\text{CNE}(i, j)$, $\text{CNW}(i, j)$, which serve as weights for the stencil.

Because a stencil may involve grid points located on different processors, inter-process communication is required. This fills ghost cells surrounding the local grid with the necessary off-process values (as previously illustrated in Figure 1). The MPI implementation for this work is listed in appendix A. In the Co-Array Fortran model data is globally accessible via load and store semantics, and the code developer is still responsible for distributing and moving the data between the parallel processes. The Co-Array Fortran implementation is shown in appendix B.

This separation of inter-process communication functionality gives us a clear picture of the coding requirements for these two programming models. In the next section we contrast these requirements with those for our Chapel implementations.

4.1 Chapel implementation

In order to illustrate some basic capabilities of Chapel, we begin with an implementation that is essentially a translation from Fortran. This eliminates the explicit inter-process communication requirements described above. We will then show how Chapel functionality enables an even more concise, expressive polymorphic version.

The first task in implementing this stencil is to define domains over which to iterate. An arithmetic domain describes the grid points in the physical domain; from it, we derive a second domain, which describes the grid points as well as space for applying the physical boundary conditions. Arrays are allocated using the latter space, while iteration is controlled by the physical space. We combine the two dimension indices into a tuple, and also define tuple-based offsets for accessing neighboring grid points, which significantly simplifies the indexing code through the use of tuple arithmetic. This version of the stencil-based matrix-vector product is shown in Figure 4. Note that the spaces containing the grid points and coefficients are defined using the global number of grid points `imt_global` and `jmt_global`, with no need to be concerned with ghost space or other issues associated with the parallel processing environment.

By viewing this operation as a reduction over the grid points in the stencil, we can express the intent of this computation even more clearly, simplifying the coding requirement as well as providing the compiler with this well known operation applied to a clearly defined data structure. In order to cleanly pose the stencil as a reduction, our Chapel implementation will generate and store the matrix elements as sets of 3×3 blocks, where each block represents the matrix coefficients for a physical grid point, as

```

1  const
2    PhysicalSpace = [1..imt_global, 1..jmt_global], // Grid points in the 2d physical
    domain.
3    AllSpace = PhysicalDomain.expand(1);           // Physical domain plus boundary.

4  var
5    X, XOUT,
6    CNW, CN, CNE, CW, CC, CE, CSW, CS, CSE // Weights in each of the nine directions.
7    : [AllSpace] real;

8  // Define neighbors:
9  const
10   NW = (-1,-1), N = (-1,0), NE = (-1,1), W = (0,-1), E = (0,1), SW = (1,-1), S = (1, 0), SE = (1,1);

11 forall i in PhysicalSpace do

12   XOUT(i) = ( CNW(i)*X(i+NW) + CN(i)*X(i+N) + CNE(i)*X(i+NE) +
13               CW(i)*X(i+W ) + CC(i)*X(i)   + CE(i)*X(i+E ) +
14               CSW(i)*X(i+SW) + CS(i)*X(i+S) + CSE(i)*X(i+SE) );

```

Figure 4: POP’s matrix-vector product in Chapel using parameterized tuple arithmetic.

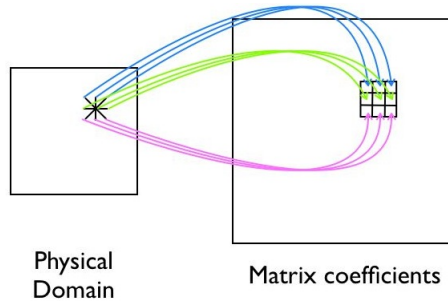
illustrated in Figure 5. That is, the nine arrays storing the matrix coefficients can be replaced with a single “array of arrays” also shown in Figure 5. Coefficients for the i^{th} linear equation are **Coeff(i)(k)** for each **k** in domain **Stencil**.

The sparse matrix-vector product may now be posed as a stencil computation, with matrix coefficients serving as weighting factors, as shown in Figure 6. The operation is addition, signified by “+”. The scope of the reduction operator is controlled by [**k in Stencil**], which is shorthand syntax for an expression-level **forall** loop.

4.2 Productivity analysis

The Fortran+MPI implementation of the matrix-vector product requires 70 lines of code, of which 43 and 21 are classified as physical and logical SLOC, respectively, and 27 whole lines of comments. Indexing into the data arrays, even when well organized, is cluttered, increasing the chance for coding errors. Though expressibility of the halo exchange is enhanced through the use of MPI derived types as well as parameterization of process ids and message tags, the code is still complicated.

The Co-Array Fortran implementation of the halo exchange requires 154 lines of code, with 104 and 84 classified as physical and logical SLOC, re-



```

1  const
2  PhysicalSpace = [1..imt_global, 1..jmt_global], // Grid points in the 2d physi-
    cal domain.
3  Stencil = [-1..1, -1..1]; // 2d 9-pt stencil domain.
4  var
5  Coeff: [PhysicalSpace][Stencil] real; // "Array of arrays" for storing ma-
    trix coefficients.

```

Figure 5: The matrix coefficients associated with each grid point are stored as 3×3 blocks in the “array of arrays” defined domain `Coeff`.

```

1  // Perform matrix-vector product:
2  forall i in PhysicalSpace do
3  XOUT(i) = + reduce [k in Stencil] Coeff(i)(k)*X(i+k);

```

Figure 6: POP’s matrix-vector product in Chapel using the reduction operator.

spectively. Complexity is even greater than with MPI, mostly attributable to the necessary explicit management of memory coherency.

The Chapel version requires only two lines of code and one line of comment. More importantly, the Chapel code clearly expresses the intent of the operation, to the benefit of the code developer as well as the compiler.

The alert reader will note that this stencil operation is directly applicable for use in many other algorithms designed to solve partial differential equations, such as finite difference methods. The version shown here is configured for a weighted stencil. A stencil which doesn’t involve weights is even simpler, eliminating the “array of arrays” of the weights (in this case matrix coefficients). Further, extending this 9-point stencil to its 27-point

analogue in three dimensions, which significantly expands the programming requirements in fragmented languages, requires only that the domains be defined using three dimensions:

```

1  const
2  PhysicalSpace = [1..m, 1..n, 1..p],
3  AllSpace      = PhysicalDomain.expand(1);
4  Stencil = [-1..1, -1..1, -1..1];

```

The polymorphic capabilities in Chapel means that the code used to perform the 2d 9-point stencil, shown previously in Figure 6, will also perform the 3d 27-point stencil. With a little extra work, we will extend this polymorphism to the 5-point stencil.

5 The 5-point stencil

Five point stencils are applied for a variety of purposes throughout POP: advection, diffusion, and other mathematical operators acting on momentum and tracer fields. Shown in Figure 7, this computation is analogous to

```

1  real (kind=dbl_kind), dimension(imt,jmt), intent(in) ::
2  & X, XOUT, ! array to operate on with 5pt operator
3  & CC,CN,CS,CE,CW ! stencil weight coefficients

4  do j=2,jmt-1
5  do i=2,imt-1
6  XOUT(i,j) = CC(i,j)*X(i,j) +
7  & CN(i,j)*X(i ,j+1) + CS(i,j)*X(i ,j-1) +
8  & CE(i,j)*X(i+1,j ) + CW(i,j)*X(i-1,j )
9  end do
10 end do

```

Figure 7: The POP 2d 5-point stencil.

the 9-point stencil described in Section 4, except that the corner elements are not included in the computation. Another differences is that, as it is used in POP, inter-process communication is a second order effect. That is, the ghost cells have already been computed using intermediate state variables, and the stencil computation also serves to generate intermediate state variables.

5.1 Chapel implementation.

At first glance the 5-point stencil might be viewed as a subset of the 9-point stencil. While true from the Fortran+MPI perspective as well as for the

Chapel implementation using parameterized tuple arithmetic (analogous to the implementation shown in Figure 4), it does not map directly to the reduction configuration we prefer since we can't define the stencil as a regular block required by the arithmetic domain. Thus far we have two options for addressing this. First, we could view the 5-point stencil as a 9-point stencil, setting corner coefficients to zero (with the associated multiplication perhaps recognized and eliminated by a compiler[14]). This has the advantage of simplicity, and could result in strong performance due to the regular blocks. However, we don't want to make such presumptions here, and more importantly, a language should be able to support this operation as well as it supports the 9-point stencil.

Our solution is to configure the stencil as a *sparse* domain, defined as a subset of the 9-point stencil arithmetic domain. While this capability is not yet implemented, we can sketch the idea, shown in Figure 8.

```

1  const
2      PhysicalSpace = [1..imt_global, 1..jmt_global] // Grid points in the physi-
    cal domain.
3      AllSpace = PhysicalSpace.expand(1);           // Physical domain plus boundary.

4      Stencil9pt = [-1..1, -1..1],
5      Stencil: sparse subdomain(Stencil9pt) = (/ (-1,0), (0,-1), (0,0), (0,1), (1,0) /);

6  var
7      Weights: [PhysicalSpace][Stencil] real;     // Matrix coefficients.

8  forall i in PhysicalSpace do
9      XOUT(i) = ( + reduce [k in Stencil] Weights(i)(k)*X(i+k);

```

Figure 8: Chapel 2d 5-point stencil.

The sparse domain creates the 5-point stencil by selecting a subset of the dense arithmetic domain which defines the 9-point stencil. As with the 9-point stencil, the reduction operator is controlled by the `Stencil` domain, providing access into the grid point data and their weights. The stencil pattern can be set several ways, including as a runtime conditional statement (shown in Figure 9), which might be useful in other situations.

5.2 Productivity analysis

The Fortran implementation of the 5-point stencil requires 7 physical SLOC, of which 5 lines are logical. (Recall that inter-process communication is not required for POP's use of this stencil.) We have discussed three possible Chapel implementations, progressing to our preference, the reduction based

```

1  const
2  Stencil: sparse subdomain(Stencil9pt) = [(i,j) in Stencil9pt]
3  if ( abs(i)+ abs(j) < 2 ) then (i,j);

```

Figure 9: Runtime conditional stencil configuration.

configuration. As with the 9-point stencil, this requires only two physical SLOC. The only difference with the 9-point (and 27-point) stencil, is the configuration of the sparse domain description of the stencil. That is, polymorphism bridges the dense and sparse domains.

The sparse domain is useful in this application, but it will prove crucial in other situations, such as operations defined on semi-structured and unstructured meshes. We've been informed that development of this capability is underway.

A different 5-point stencil routine is used for computing the horizontal part of momentum advection in POP. It is more complex than the stencil show above, so we don't include it here. However, it's worth mentioning that it is applied to each of the vertical levels within a nested loop. The levels are not coupled (for this calculation), so the data could be configured so that the computation could operate throughout the levels in a single data structure. The lack of memory layout constraints by Chapel provides the compiler with the flexibility to configure out data such that this operation is most advantageous. Where levels must be distinguished, a separate domain could be defined. In fact, we could envision a global view implementation that might express a task based parallelism, one task per level, which could be exploited by particular architecture features. However we temper our choice here with the concession that a full analysis of the POP model would be needed before settling on this (or any other) data structure.

6 A Brief Note Regarding Performance Expectations

An expressive language that performs poorly will not satisfy the overall productivity goals of most HPC users. In a sense our work here forms the basis for a study on the performance potential of Chapel. We look forward to developing that work, as well as tracking the performance of this and other codes as compiler development progresses.

The global expression of an algorithm can provide meaningful flexibility

to the compiler, enabling a basis for strong use of asynchronous communication, sharing data as it is needed, taking advantage of architecture specific runtime capabilities, etc. For example, a reduction over a domain provides a strong description of the intent of the stencil computation. The fragmented view compels the use of halos (called ghosts in POP), injecting synchronization in order to manage the transfer of data. Although we can envision a similar implementation by a Chapel compiler, this is not an inherent characteristic. And as we saw, the MPI model as used by POP provides the user with control over coherency, so, for example, no inter-process communication is necessary for the 5-point stencil. How might a global view language compiler then avoid unnecessary communication[¶]? We intend to investigate these issues.

7 Conclusions and Future Work

Using a well-known and widely used scientific application program, we have demonstrated how Chapel can express some crucial computations. Chapel lets the code developer express not only these important computations but also their variants in a clear, concise syntax. This claim is strengthened by the polymorphic capability of Chapel illustrated herein that demonstrates the value of data structure independence from computation. Further, while we don't believe a simple comparison of the number of lines of code captures the essence of productive code development, it is significant that a full direct translation of POP from MPI to Chapel would eliminate approximately 6,000 lines of code. We've also shown a significant code development advantage of the Chapel implementation over the prototyped Co-Array Fortran implementation of the halo exchange.

As previously stated, this report is not intended as a tutorial for creating a Chapel version of POP. POP exists, with a base of 36,702 lines of code, so without a significantly compelling reason for doing so, we doubt the POP developers would even consider such an implementation. Yet this points precisely to the purpose of this work: evaluating Chapel capabilities in order to determine if such compelling reasons exist. To our knowledge, this paper represents the first attempt to do so within the context of computations actually used in large scale scientific applications.

We are actively investigating the use of Chapel, as well as Fortress and X10, for other classes of computations. More interesting (and more challenging!) are our investigations into how Chapel constructs might influence

[¶]This issue has been addressed in ZPL[7] so we expect it to be addressed in Chapel.

the development and choice of algorithms in posing computational science experiments.

Acknowledgements

We are grateful to Brad Chamberlain, the Technical Lead of the Chapel project, for invaluable discussions, advice, and code review. He has taken seriously our many discussions regarding our perceptions of the value of existing as well as potential functionality in the Chapel language. We extend this gratitude to all members of the Chapel team who, through Brad, aided our efforts. We look forward to continued interactions. We are also thankful to DARPA for the vision to explore these new languages which will be necessary as we enter the ExaFLOP computing regime.

References

- [1] E. Allen, D. Chase, J. Hallet, V. Luchangco, J. Maessen, S. Ryu, G. L. Steele Jr, and S. Tobin-Hochstadt. The Fortress Language Specification, version 1.0. β . Technical report, Sun Microsystems, Inc., 2007.
- [2] M.B. Blackmon, B. Boville, F. Bryan, R. Dickinson, P. Gent, J. Kiehl, R. Moritz, D. Randall, J. Shukla, S. Solomon, G. Bonan, S. Doney, I. Fung, J. Hack, E. Hunke, J. Hurrell, and et al. The Community Climate System Model. *BAMS*, 82, 2001.
- [3] B.L.Chamberlain. *The Design and Implementation of a Region-Based Parallel Programming Language*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 2001.
- [4] B.L. Chamberlain, D.Callahan, and H.P. Zima. Parallel programming and the Chapel language. *International Journal on High Performance Computer Applications*, 21(3):291–312, 2007.
- [5] Brad Chamberlain. Chapel development team, Private communication, 2005-07.
- [6] P. Charles, C. Donawa, K. Ebcioğlu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA)*, October 2005.

- [7] Sung-Eun Choi and Lawrence Snyder. Quantifying the effects of communication optimizations. In *IEEE International Conference on Parallel Processing*, 1997.
- [8] Cray, Inc. Cray MTA-2 Programmer’s Guide. S-2320-10, 2005.
- [9] Cray, Inc. Chapel Language Specification 0.750. <http://chapel.cs.washington.edu>, 2007.
- [10] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), 1998.
- [11] DARPA. High Productivity Computing Systems program. <http://highproductivity.org>, 1999.
- [12] R.E. Diaconescu and H.P. Zima. An Approach to Data Distribution in Chapel. *International Journal on High Performance Computer Applications*, 21(3), 2007.
- [13] S. J. Dietz, D. Callahan, B.L. Chamberlain, and L. Snyder. Global-view abstractions for user-defined reductions and scans. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2006.
- [14] S. J. Dietz, B.L. Chamberlain, and L. Snyder. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the ACM International Conference on Supercomputing*, 2001.
- [15] P.A. Agarwal et al. Cray X1 evaluation status report. In *Proceedings of the 46th Cray User Group Conference*, 2004.
- [16] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference: Volume 2 - The MPI-2 Extensions*. The MIT Press, 1998.
- [17] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.
- [18] High Performance Fortran Forum. High Performance Fortran language specification. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1993.

- [19] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su, and K. Yelick. Titanium language reference manual. Technical Report UCB/EECS-2005-15, University of California, Berkeley, 2005.
- [20] P.W. Jones, P.H. Worley, Y. Yoshida, J. B. White, and J. Levesque. Practical performance portability in the parallel ocean program (POP). *Concurrency and Computation: Experience and Practice*, 2004.
- [21] R.J. Murray. Explicit generation of orthogonal grids for ocean models. *Journal of Computational Physics*, 251, 1996.
- [22] R.W. Numrich and J.K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998.
- [23] Parallel Ocean Program. (POP). <http://climate.lanl.gov/Models/POP>.
- [24] University of Southern California Center for Software Engineering. CodeCountTMtoolset. <http://sunset.usc.edu/research/CODECOUNT>, 1998.
- [25] R.D. Smith and S. Kortas. Curvilinear coordinates for global ocean models. Technical Report LA-UR 95-1146, Los Alamos National Laboratory, 1995.
- [26] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference: Volume 1 - 2nd Edition*. The MIT Press, 1998.
- [27] UPC. Consortium, UPC Language Specification. May 31 2005.

A POP’s MPI halo exchange

```

1 !   fill buffers and send east-west boundary info
2   i = 1
3   do n=1,num_ghost_cells
4     do j=jphys_b,jphys_e
5       buffer_east_snd(i)=XOUT(ipphys_e+n-num_ghost_cells,j)
6       buffer_west_snd(i)=XOUT(ipphys_b+n-1,j)
7       i=i+1
8     end do
9   end do
10  call MPI_ISEND(buffer_east_snd,
```

```

11      &          buf_len_ew, MPI_DOUBLE_PRECISION, nbr_east,
12      &          mpitag_wshift, MPI_COMM_OCN, request(1), ierr)

13      call MPI_ISEND(buffer_west_snd,
14      &          buf_len_ew, MPI_DOUBLE_PRECISION, nbr_west,
15      &          mpitag_eshift, MPI_COMM_OCN, request(2), ierr)

16 !      receives east-west boundary info and copy buffers into ghost cells

17      call MPI_RECV(buffer_west_rcv,
18      &          buf_len_ew, MPI_DOUBLE_PRECISION, nbr_west,
19      &          mpitag_wshift, MPI_COMM_OCN, status, ierr)

20      call MPI_RECV(buffer_east_rcv,
21      &          buf_len_ew, MPI_DOUBLE_PRECISION, nbr_east,
22      &          mpitag_eshift, MPI_COMM_OCN, status, ierr)

23      call MPI_WAITALL(2, request, status_wait, ierr)

24      i = 1
25      do n=1,num_ghost_cells
26          do j=jphys_b,jphys_e
27              XOUT(n,j) = buffer_west_rcv(i)
28              XOUT(ipphys_e+n,j) = buffer_east_rcv(i)
29              i=i+1
30          end do
31      end do

32 !      send north-south boundary info

33      call MPI_ISEND(XOUT(1,jphys_e+1-num_ghost_cells), buf_len_ns,
34      &          MPI_DOUBLE_PRECISION, nbr_north,
35      &          mpitag_sshift, MPI_COMM_OCN, request(1), ierr)

36      call MPI_ISEND(XOUT(1,jphys_b), buf_len_ns,
37      &          MPI_DOUBLE_PRECISION, nbr_south,
38      &          mpitag_nshift, MPI_COMM_OCN, request(2), ierr)

39 !      receive north-south boundary info

40      call MPI_RECV(XOUT(1,jphys_e+1), buf_len_ns,
41      &          MPI_DOUBLE_PRECISION, nbr_north,
42      &          mpitag_nshift, MPI_COMM_OCN, status, ierr)

43      call MPI_RECV(XOUT(1,1), buf_len_ns,
44      &          MPI_DOUBLE_PRECISION, nbr_south,
45      &          mpitag_sshift, MPI_COMM_OCN, status, ierr)

46      call MPI_WAITALL(2, request, status_wait, ierr)

```

B POP's Co-Array Fortran halo exchange

```

1      if (first) then

```

```

2     left_cell_ready = .false.
3     right_cell_ready = .false.
4     top_cell_ready = .false.
5     bot_cell_ready = .false.
6     left_cell_done = .false.
7     right_cell_done = .false.
8     top_cell_done = .false.
9     bot_cell_done = .false.
10    first = .false.
11    call sync_images()
12    endif

13 !   Perform computation (not shown here).

14 !   determine neighboring cell indices.

15    if(me(1).ne.NPROC_X)then
16        me1p1 = me(1)+1
17    else
18        me1p1 = 1
19    endif

20    if(me(1).ne.1)then
21        me1m1 = me(1)-1
22    else
23        me1m1 = NPROC_X
24    endif

25    if(me(2).ne.NPROC_Y)then
26        me2p1 = me(2)+1
27    else
28        me2p1 = 1
29    endif

30    if(me(2).ne.1)then
31        me2m1 = me(2)-1
32    else
33        me2m1 = NPROC_Y
34    endif

35 !       inform neighbors that I am ready for my halo
36 !       points to be updated

37     right_cell_ready[me1m1,me(2)] = .true.
38     left_cell_ready[me1p1,me(2)] = .true.
39     top_cell_ready[me(1),me2m1] = .true.
40     bot_cell_ready[me(1),me2p1] = .true.

41 !   update left and right ghost cell boundaries.
42 !   wait until left and right neighbors are ready

43     do while ((.not. left_cell_ready) .or. (.not. right_cell_ready))
44     enddo

45     left_cell_ready = .false.
46     right_cell_ready = .false.

```

```

47     do n=1,num_ghost_cells
48         do j=jphys_b,jphys_e
49             XOUT(ipphys_e+n,j)[me1m1,me(2)] = XOUT(ipphys_b+n-1,j)
50             XOUT(
51                 n,j)[me1p1,me(2)] =
52                 XOUT(ipphys_e-num_ghost_cells+n,j)
53         end do
54     end do

54     call sync_memory()
55 !     inform left and right neighbors that I am finished

56     left_cell_done[me1p1,me(2)] = .true.
57     right_cell_done[me1m1,me(2)] = .true.

58 !     wait until left and right neighbors have completed updating
59 !     my ghost region

60     do while ((.not. left_cell_done) .or. (.not. right_cell_done))
61     enddo

62     left_cell_done = .false.
63     right_cell_done = .false.

64 !     update top and bottom ghost cell boundaries.

65 !     wait until top and bottom neighbors are ready

66     do while ((.not. bot_cell_ready) .or. (.not. top_cell_ready))
67     enddo

68     bot_cell_ready = .false.
69     top_cell_ready = .false.

70     do n=1,num_ghost_cells
71         do i=1,imt
72             XOUT(i,jphys_e+n)[me(1),me2m1] = XOUT(i,jphys_b+n-1)
73             XOUT(i,
74                 n)[me(1),me2p1] =
75                 XOUT(i,jphys_e-num_ghost_cells+n)
76         end do
77     end do

77     call sync_memory()

78 !     inform top and bottom neighbors that I am finished

79     top_cell_done[me(1),me2m1] = .true.
80     bot_cell_done[me(1),me2p1] = .true.

81 !     wait until top and bottom neighbors have completed updating
82 !     my ghost region

83     do while ((.not. bot_cell_done) .or. (.not. top_cell_done))
84     enddo

85     bot_cell_done = .false.

```

86 `top_cell_done = .false.`
