Exploring the Performance Potential of Chapel in Scientific Computations [*]

Richard F. Barrett, Sadaf Alam, and Stephen W. Poole
Oak Ridge National Laboratory
Oak Ridge, TN 37931

Presented at Cray User Group, Helsinki, Finland, May 7, 2008

Abstract

Languages are being designed that simplify the tasks of creating, extending, and maintaining scientific application specifically for use on parallel computing architectures. Widespread adoption of any language by the high performance computing (HPC) community is strongly dependent upon achieved performance of applications. A common presumption is that performance is adversely affected as the level of abstraction increases. In this paper we report on our investigations into the potential of one such language, Chapel, to deliver performance while adhering to its code development and maintenance goals. In particular, we explore how the unconstrained memory model presented by Chapel may be exploited by the compiler and runtime system in order to efficiently execute computations common to numerous scientific application programs. Experiments, executed on a Cray X1E, AMD dual-core, and Intel quad-core processor based systems, reveal that with the appropriate architecture and runtime support, the Chapel model can achieve performance equal to the best Fortran/MPI, Co-Array Fortran, and OpenMP implementations, while substantially easing the burden on the application code developer.

1 Introduction

The wealth of emerging high performance computing architectures are of great interest to the scientific computing community. Features include multi-core, heterogeneous processors, complex memory hierarchies, and heterogeneous interconnects. Unfortunately current programming models are not capable of fully exploiting the enormous performance potential of these architectures[1, 17, 23]. Current models are designed to provide access to low level runtime system mechanisms in order to achieve performance. Although this approach works well on the relatively homogenous architectures that currently define most high performance computing (HPC) systems, their low-level approach over constrains the runtime capabilities of these new architectures.

It is a common belief that a higher level of abstraction necessarily imposes penalties on runtime performance. Yet if viewed from a different perspective, this need not be the case. In fact the very lack of specification for *how* a computation should be structured and executed provides the compiler and runtime system with opportunities for exploiting advanced architectural capabilities that may be ruled out through the use of lower level models.

The Chapel programming language[8] is designed with a higher-level of abstraction. A component of the Cray Cascade project, and funded by the DARPA High Productivity Computing Systems (HPCS) program[14], the goal of Chapel is to satisfy the components of the term "productivity" as they relate to user requirements. Although productivity is often erroneously viewed as simply a code development issue, the characteristics of a productive language are programmability, performance, portability, and robustness. That is, while programs should be easier to write relative to existing methods, such programs must also yield strong performance across a variety of computing architectures, and these characteristics must be easier to maintain as an application is developed and extended.

[*]This research has been supported by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), and used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

In this report we explore how Chapel may be used to drive performance strategies relative to algorithmic expressions rather than the finer-grained loop level computations. Specifically, we examine Chapel's ability to configure and execute finite difference stencils, computations common to a broad set of scientific applications. Because a high performance, parallel processing compiler is not yet available, our experiments simulate the kinds of configurations enabled by Chapel's higher level abstraction using lower level language implementations. Implementations are constructed using Fortran and MPI[25, 18], Co-Array Fortran (CAF[21]), and OpenMP[13]. The construction of these experiments is based on the idea that Chapel's unconstrained memory model presents significant opportunities for exploiting architectural characteristics and applying flexible programming methodologies, including hybrid techniques.

This report is organized as follows: after a brief discussion of related research, we describe the architectures upon which scientific applications are expected to execute and achieve highest performance. This motivates the discussion of the Chapel programming language. After a brief overview of the finite difference approach, we examine important computations used in the algorithm, comparing and contrasting implementations written using the MPI message passing model, Co-Array Fortran, OpenMP, and Chapel. Next we describe and discuss experiments constructed to illustrate the performance potential of the Chapel implementations. Lastly we offer our conclusions and discuss our future research.

1.1 Related research

The message passing model using MPI is the predominant method for creating scientific application programs for parallel processing architectures. PGAS languages have been developed in order to address some of the difficulties perceived with MPI-based programs. Although we examine only Co-Array Fortran from that set in this report, interested readers should be familiar with other alternatives, such as Unified Parallel C (UPC[26]), a set of extensions to the C programming language; and Titanium[19], a Java-based language. OpenMP, also used herein, is often combined with MPI in order to span regions of physically shared memory. Two other notable global-view language development efforts are also underway: Fortress[2] endeavors to present a mathematically based syntax to the code developer. X10[10] extends Java with support for parallel programming. We are investigating these languages in a manner similar to that described in this report.

2 Architectures

The stated trend of all major processor vendors is based on heterogeneous multi-core processors, increased memory hierarchies, with larger physical and logical memory spaces, and hierarchical heterogeneous interconnects. Processors will consist of scalar, vector, graphics processor unit (GPU), field-programmable gate arrays (FPGA), multi-threading, and perhaps as yet unspecified processor accelerators. Memory models and inter-process communication protocols promise to be even more hierarchical and flexible.

2.1 Cray X1E

The X1E is a massively parallel processing (MPP) system, hierarchical in processor, memory, and network design. The basic building block of the X1E is a compute module, each consisting of four multi-chip modules (MCM), 16 GBytes of memory, routing logic, and external connectors. Two multi-streaming processors (MSP) occupy an MCM. An MSP, shown in Figure 1, consists of four single-streaming processors (SSP), each with two 32-stage 64-bit floating point vector units and one 2-way super-scalar unit. An SSP uses two clock frequencies: 1.13 GHz for the vector units, and 400 MHz for the scalar units.

The interconnect functions as an extension of the memory system, offering each node direct access to memory on other nodes at high bandwidth and low latency. Thus for partitioned global address space languages like CAF, each processor can directly address memory on any other node. These remote memory accesses are issued directly from the processors as load and store instructions, transparently executed over the X1E interconnect to the target processor, bypassing the local cache. Access costs are listed in Table 1. Both scalar and vector loads are blocking primitives, limiting the ability of the system to overlap communication and computation.

The X1E MSP can be viewed as a heterogeneous (vector and scalar) multi-core (4 SSP per MSP) hierarchical (4 MSP per module) processor and network design, and thus is representative of the future architectures that we anticipate will soon dominate the high performance scientific computing community. In particular,

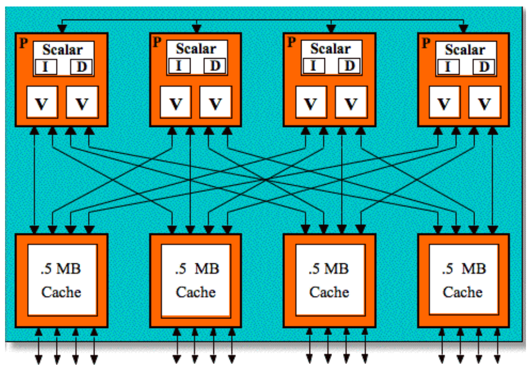


Figure 1: The Cray X1E MSP. (Image courtesy of Cray, Inc.)

Memory location	*Relative access time*
D-cache	1X
E-cache	2X
Local (node) memory	7X
Remote (off node) memory	10X-32X

Table 1: Cray X1E memory access costs.

the X1E provides support for the kinds of activities necessary for effective execution of Chapel constructs on multi-core processor based systems.

2.2 Multi-core micro-processor systems

Two multi-core processor based systems were used in this study. The first system consists of eight dual-core AMD 8216 Opteron sockets, spanning 32 GBytes of shared memory. Each core, with a clock speed of 2.4 GHz, has a 1 MByte L2 cache. The second, an Intel Clovertown system, consists of two Xeon 5160 quad-core sockets, spanning 4 GBytes of shared memory. Each socket, with a clock speed of 2.4 GHz, shares a 4 MByte L2 cache. A detailed analysis of this kind of system for use by scientific applications is presented in [24].

3 Chapel Overview

MPI and CAF provide a "local-view" of parallel computation in that they require the code developer to explicitly manage both the interaction of the parallel processes and the overall data layout. In contrast, Chapel provides a "global-view" of the computation and associated data[1], illustrated in Figure 2. In order to provide an easier means for writing code for execution on parallel processing architectures.

This approach has been tried before, and the results were unsatisfactory for the majority of the scientific computing community. Chapel is attempting to overcome this history by, among other things, providing mechanisms for defining and distributing global data structures, called *domains*. Domains are first class objects, constructs which provide the code developer with a means for configuring data structures. *Distributions* qualify a domain, defining how data are decomposed across the parallel processes. 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.

In particular, the global-view model eliminates the syntactic distinction between local and remote memory access that is found in local-view languages. Moreover, Chapel imposes no constraint upon the compiler for the manner in which multi-dimensional arrays are laid out in memory. (Fortran and C, for example, require

[1]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.

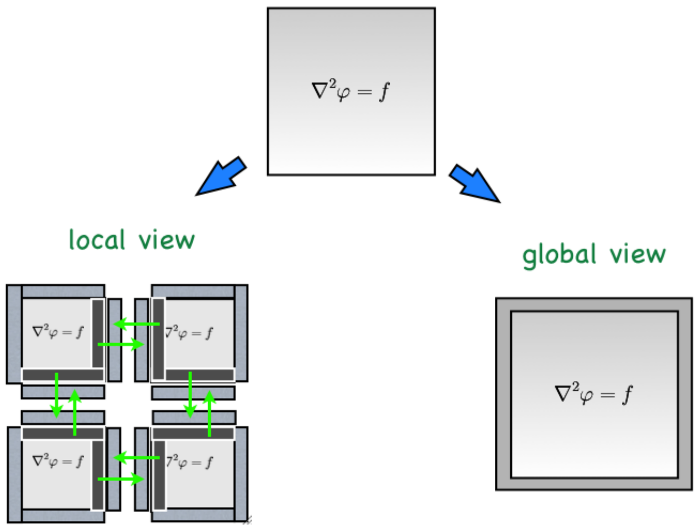


Figure 2: Local-view parallel vs. global-view programing model.
Consider a partial differential equation (here Poisson's Equation) defined on a two-dimensional domain The local-view configuration for applying a solution algorithm on a parallel processing computer is shown on the right. Here the code 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 local model) may then surround with space for the physical boundary conditions. Although the language may provide semantics for conveying information regarding parallelism for a particular problem (Chapel does[15]), the code developer is not responsible for distributing and sharing data among the parallel processes.

column-major and row-major ordering, respectively.) In this section we discuss issues and opportunities presented by Chapel's unconstrained memory model.

3.1 Remote memory access

There is a presumption that particular data decompositions are inherent for particular algorithms, independent of the architecture and its runtime system capabilities. However, this presumption is based on language and model constraints rather than algorithmic and architecture characteristics.

When viewed from the perspective of the algorithm and its computations, inter-process communication simply presents another level of memory hierarchy. However, this perspective is not clear when using local-view models since the user must explicitly switch mechanisms to access data on remote processes. For example, when using MPI, computation is interrupted while user-defined message passing functionality moves data between the parallel processes, often requiring intermediate buffering. Although asynchronous schemes can be configured, not all architectures support this capability, often resulting in degradation of performance. CAF presents a load/store model for accessing data on remote processes, but this too requires explicit coding by the programmer, preventing the language from injecting any sense of intent of the computation to the compiler.

The global-view model allows the compiler and runtime system to recognize the intent of a computation and thus organize computation and communication accordingly. For example, in Chapel, a block decomposition of the data may not be the optimal distribution on all architectures. Chapel provides other distribution strategies that could be easily configured, such as a graph-partitioning strategy. More interesting to us is an option under consideration that would call for a distribution of the data across the parallel processes, but leaves the decision to the compiler and runtime system[9].

3.2 Local memory

Fortran, C, C++, and other languages common to scientific computing organize the storage of multi-dimensional arrays in some well-defined fashion. This compels the code developer to order computations so that data encountered in computations map effectively to the processor architecture. Also, the low level coding required of these models can result in unnecessary constraints on compiler optimization strategies.

Chapel leaves organization of storage unspecified. Further, it defines *tuples*, allowing the indices of multi-dimensional arrays to be combined. Beyond the coding convenience, this provides a flexible model for the internal ordering of data that could be exploited in concert with the remote memory access requirements of a computation. Although not investigated herein, we intend to explore opportunities in this realm. S_N transport as implemented in Sweep3d[20], where waves traverse the physical domain, is an algorithm that could take advantage of this flexibility[3].

The Chapel language specification[12] is at version 0.775. A prototype compiler (pre-release version 0.7) has been provided to a small group of programmers who are gaining experience and providing feedback to the Chapel developers.

4 Finite Difference Stencils

A broad range of physical phenomena in science and engineering can be described mathematically using partial differential equations. Determining the solution of these equations on computers is often accomplished using finite differencing methods. The algorithmic structure of these methods maps naturally to the data parallel programming model. However, implementing these algorithms effectively is becoming increasingly difficult as HPC architectures become more complex.

Finite difference methods are mathematical techniques for approximating derivatives or a differential operator by replacing the derivatives with some linear combination of discrete function values. An example of one such differential equation is Poisson's equation

$$-(u_{xx} + u_{yy}) = f(x, y), \tag{1}$$

perhaps defined on

$$\Omega = [0, N] \times [0, N], \text{with } u = 0 \text{ on } \delta\Omega.$$

(This equation is often written as $\nabla^2\varphi = f$.) We discretize Ω with resolution $1/h$, resulting in $(N/h) + 1 = n$ grid points in each dimension. During an iteration, each grid point is updated as a function of the current value of it and some combination of its neighbors. This computation is often described as applying a *stencil* to each point of the grid (as illustrated in Figure 3).

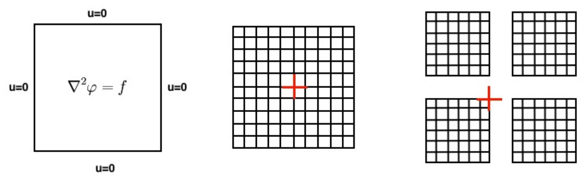


Figure 3: Solving the 2-D Poisson Equation using a 5-point difference stencil.
The figure on the left shows the Poisson Equation defined on a continuous domain, with Dirichlet boundary conditions. The center figure shows the domain discretized, with a 5-point difference stencil, in red. The figure on the right shows the domain divided up as blocks for mapping to a parallel processing computer.

When mapping this algorithm to a parallel processing computer, the typical data decomposition strategy divides the domain into blocks, assigning one or more block to each process. This creates artificial interior boundaries, along which each parallel process must access off-process data (called ghosts, shadows, or halos, etc) in order to compute over the stencil[2]. Our implementations using Fortran with MPI, Co-Array Fortran, OpenMP, and Chapel are described in this section.

[2]A potential non-trivial difference among these implementations is their requirement for user-allocated memory for storing inter-process boundary data. However, the compiler may allocate buffer space for a pre-fetching scheme.

```
real, dimension( nrows_loc+2, ncols_loc+2 ) :: grid1, grid2

call exchange_boundary ( ... )

do j = 2, ncols_loc-1
    do i = 2,nrows_loc-1
        grid2(i,j) = (                                    &
                        grid1(i-1,j) +                    &
            grid1(i,j-1) + grid1(i,j)   +  grid1(i,j+1) +  &
                        grid1(i+1,j) ) / 5.0
   end do
end do
```

Figure 4: Fortran 5-point stencil in 2-D.

4.1 Fortran-MPI

The MPI specification provides several methods for moving data from one parallel process to another. We abstract the mechanics of the difference stencil halo exchange into procedure `exchange_boundary`. Our Fortran implementation is shown in Figure 4. More sophisticated implementations might enable latency hiding, at the expense of increased coding complexity.

For our experiements, we first post the non-blocking receives (`MPI_Irecv`), followed by the non-blocking sends, followed by a loop over `MPI_Waitany`. This increases the opportunity for executing in "expected message" mode, which can reduce intermediate buffering.

4.2 Co-Array Fortran

The computational loop for our CAF implementation, shown in Figure 5, makes no distinction between local and remote array accesses. This "load-it-as-you-need-it" style provides flexibility to the compiler and

```
real, dimension( nrows_loc, ncols_loc )[*] :: grid1, grid2

call sync_team ( neighbors )

do j =  1, lcols
   do i =  1, lrows
      left     = grid1(ii(i,j-1),jj(i,j-1))[img_loc(i,j-1)]
      top      = grid1(ii(i-1,j),jj(i-1,j))[img_loc(i-1,j)]
      center   = grid1(i,j)
      bottom = grid1(ii(i+1,j),jj(i+1,j))[img_loc(i+1,j)]
      right    = grid1(ii(i,j+1),jj(i,j+1))[img_loc(i,j+1)]

      grid2(i,j) = ( left+top+center+bottom+right ) / 5.0
   end do
end do
```

Figure 5: CAF 5-point stencil in 2-D.

runtime system with regard to pre-fetching, pipelining, and other scheduling mechanisms, and therefore should provide the best chance for hiding remote image load latencies. Viewed another way, this places all responsibility for load/store scheduling on the compiler, yet no specific information regarding inter-image data sharing is available until runtime. Further, as discussed in Section 5.3, compilers inject overhead due to the presence of the co-array bracket notation in spite of the fact that $O(N^2)$ loads will be local to the image compared with $O(N)$ off-image. The indirect addressing within arrays will probably also degrade performance.

We will employ two additional strategies that could be configured by a CAF code developer, though it is our contention that these implementations are not the "natural" use of CAF and thus do not conform to the intended model, an issue discussed in [6]. However, these approaches could be employed by a Chapel compiler.

First, analogous to the MPI model, the compiler could simply recognize that the block decomposition creates regular boundaries that must be shared among the parallel processes. It could then exchange boundaries in bulk using CAF remote loads of arrays segments into local process ghost zones. The code fragment implementing this strategy is shown in Figure 6. For convenience we will subsequently refer to this as the

```
real, dimension( nrows_loc+2, ncols_loc+2 )[*] :: grid1, grid2

call sync_team(neigh)

if ( neigh(south) /= my_image) &   ! Get south boundary.
  grid1(lrows+2,2:lcols+1) = grid1(2,2:lcols+1)[neigh(south)]

if ( neigh(north) /= my_image) &   ! Get north boundary.
  grid1(1,2:lcols+1) = grid1(lrows+1,2:lcols+1)[neigh(north)]

if ( neigh(west) /= my_image) &   ! Get west boundary.
  grid1(2:lrows+1,1) = grid1(2:lrows+1,lcols+1)[neigh(west)]

if ( neigh(east) /= my_image) &   ! Get east boundary.
  grid1(2:lrows+1,lcols+2) = grid1(2:lrows+1,2)[neigh(east)]

call sync_team(neigh)

! Begin computation: all data now local.
```

Figure 6: Shmem model.

CAF-Shmem version.

The compiler could recognize that off-image data is needed only for grid points along inter-image boundaries, and therefore set up several separate computational loops: one operating solely on the inner grid (no remote data accesses required), and one loop along each boundary. The latter are intended to inform the compiler of the regular data access patterns. This also eliminates the need for the co-array notation for each array point, required for the CAF version. A code fragment illustrating the idea is shown in Figure 7. For

```
real, dimension( nrows_loc, ncols_loc )[*] :: grid1, grid2

if ( neigh(north) /= my_image ) then
   do j = 2, lcols-1
       grid2(1,j) =                                        &
                         ( grid1(lrows,j)[neigh(north)] +  &
            grid1(1,j-1)+ grid1(1,j) + grid1(1,j+1) +      &
                          grid1(1,j) ) / 5.0
   end do
else
   do j = 2, lcols-1
      grid2(1,j) =                                         &
           ( grid1(1,j-1) + grid1(1,j) + grid1(1,j+1) +    &
                           grid1(2,j) ) / 5.0
   end do
end if
```

Figure 7: Segmented model.
The computation across the north boundary of the local grid points, interleaving local and remote data accesses.

convenience, we will subsequently refer to this as the CAF-Segmented version.

4.3 OpenMP

OpenMP compiler directives let us easily create a multi-processor threaded implementation from our serial Fortran implementation. In this case, we simply surround the source code in Figure 4 with a parallel `DO` directive. In order to prevent thread startup costs from being included, we surround this code (and some administrative code) with a `PARALLEL` region, and precede the timed computation with a barrier.

4.4 Chapel

Although we could simply create a syntactic translation of the Fortran implementation to Chapel, we can better express the intent of the computation by viewing the stencil as a reduction. In this section we describe this approach. (A detailed discussion of these implementations and their use in scientific applications is presented in [5, 6].) We begin with the 9-point stencil.

An arithmetic domain describing the grid points in the physical space is defined. From it, we derive a second domain, which adds space for applying boundary conditions. Arrays are allocated using this boundary space, while iteration is controlled by the physical space. We combine the two dimension indices into a tuple. By viewing the stencil operation as a reduction over a set of grid points, we can more clearly express the intent of this computation. In addition to producing more readable code, this conveys to the compiler the intent of the stencil computation within the context of the data structure. The code for this implementation is shown in Figure 8.

```
const  PhysicalSpace :
   domain(2) distributed(Block) = [ 1..m, 1..n ],
      AllSpace = PhysicalSpace.expand(1);

const  Stencil9pt = [ -1..1, -1..1 ];

var Grid1, Grid2 : [AllSpace] real;

forall i in PhysicalSpace do
   Grid2(i) = ( + reduce [ k in Stencil ] Grid1(i+k) ) / 9.0;

Note: The notation [  k in Stencil ] is equivalent to
     forall k in Stencil do
```

Figure 8: Chapel 9-point stencil in 2-D.

The 5-point stencil can be viewed as a subset of the 9-point stencil when implementing it using Fortran+MPI, CAF, and an analogous Chapel translation. However, it does not map directly to the Chapel reduction configuration we prefer since we cannot define the stencil as a regular block required by the arithmetic domain. We could use the 9-point stencil domain by setting corner coefficients to zero (with the associated multiplication perhaps recognized and eliminated by a compiler[16]). This has the advantage of simplicity, and could result in strong performance associated with regular blocks. However, we do not want to make such assumptions here, and more importantly, a language should be able to support this operation as well as it supports the 9-point stencil.

Chapel does provide such support by letting us configure the stencil as a *sparse* domain, defined as a subset of the 9-point stencil arithmetic domain, shown in Figure 9. The sparse domain creates the 5-point

```
// Same declarations as Figure 8, plus:

const
   Stencil: sparse subdomain( Stencil9pt ) =
     ( (-1,0), (0,-1), (0,0), (0,1), (1,0) );

forall i in PhysicalSpace do
   Grid2(i) = ( + reduce [ k in Stencil ] Grid1(i+k) / 5.0;
```

Figure 9: Chapel 5-point stencil in 2-D.

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, which might be useful in other situations.

5 Experimental Results

In this section we present results of experiments run on the architectures described in Section 2. From these we discuss the performance potential of the Chapel language global-view abstraction.

5.1 Weak Scaling Performance Evaluation

Experiments designed to illustrate weak scaling for large processor counts were run on Phoenix, the 1,024 MSP Cray X1E located at Oak Ridge National Laboratory[22]. The largest local grid dimension shown in these results is 8000, which consumes about half of the memory available for each MSP. However, grid dimensions that consumed practically all available memory (14,000) showed the same performance characteristics.

Our stencil implementations are straightforward, clearly exposing the work required for parallel processing. The computational workload is captured in a doubly nested loop, which maps well to the X1E MSP processor, where it operates entirely in vector and multi-stream mode[3]. All experiments were run in MSP mode under the UNICOS operating system, release 3.1.0.7, within the default programming environment, version 5.5.0.1.

The performance of the various 5-point stencil implementations are shown in Appendix A. For small local grid dimensions, CAF-Shmem and CAF-Segmented significantly out perform the CAF version, which in turn out performs the MPI version. We attribute this to three factors. First, the co-array protocol for loading (and storing) data from and to remote images takes advantage of the X1E inter-processor communication infrastructure. Second, the CAF version incurs the co-array cost for local data. Third, message latencies dominate the MPI implementation.

However, as the local grid dimension increases, the cost of the co-array semantic for local references in the CAF version increases linearly with $O(N^2)$ computational intensity. It is further limited by the indirect addressing, and thus its performance stays flat. CAF-Shmem and CAF-Segmented do not incur the unnecessary co-array cost because we have explicitly incorporated knowledge of their inter-image data sharing requirements, and thus their performance increases with the computational intensity. The fixed cost (latency) of the MPI version is amortized across the larger message sizes, and thus its performance increases as well.

The 9-point stencil introduces up to four new (diagonal) partner processes, each contributing only a single grid point. This should not present any problems to the CAF version since its simply another load. It also plays to a particular strength of the message passing model, since with a little attention to message coordination, the programmer can avoid increasing the number of messages required for the 5-point stencil, albeit at the expense of an extra inter-neighbor synchronization point: complete the boundary transfer with north and south neighbors, then exchange boundaries with east and west neighbors. The first exchange places the single point from each diagonal neighbor onto the horizontal neighbor, which is then properly shared in the east-west exchange. (Of course the order could be east-west, then north-south.) This method is also applied in the CAF-Shmem implementation.

The performance of the four 9-point stencil implementations is shown in Appendix B. For the smallest grid size, the relative performance stays about the same as with the 5-point stencil, although the CAF-Segmented and CAF-Shmem versions are about even for the middle sizes. All implementations outperform their 5-point counterparts, due to the increased computation relative to the amount of communication.

For larger grid sizes, unlike the 5-point stencil, the MPI version becomes the best performer. We expected it to outperform the CAF-Segmented version since it (CAF-Segmented) must load the diagonal elements as four single coefficients, each from a different image. As with MPI, the CAF-Shmem version requires two synchronization points, between pairs of processes/images; we speculate that the asynchronous nature of MPI outperforms the two calls to `SYNC_TEAM`, a notion we will examine more closely in future work.

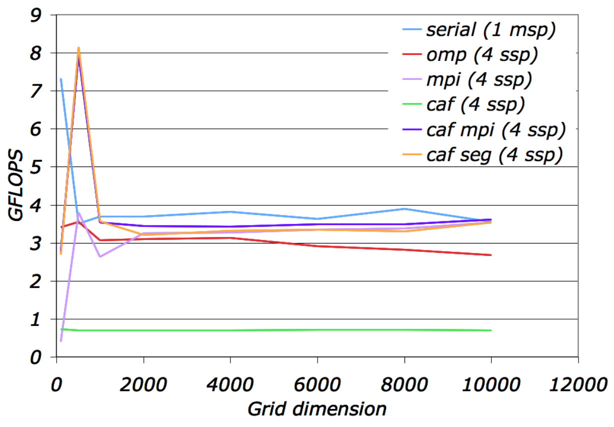
5.2 Multi-core performance evaluation

Performance evaluation of a parallel program begins with examining performance on a single processor. However, the definition of a "single" processor depends on the view presented to the programmer. For example, the X1E MSP can be programmed as if it were a single processor, even using local-view language models, yet it consists of four SSPs; further, the local-view can also be narrowed to a single SSP. Similarly a multi-core micro-processor can be programmed as a collection of its CPUs or at the individual CPU level. These local views are then tied together using MPI or some other protocol. Although Chapel would hide these details from the programmer, the compiler would still have to address the single processor view at some level, and therefore we examine the performance opportunities afforded by various mechanisms.

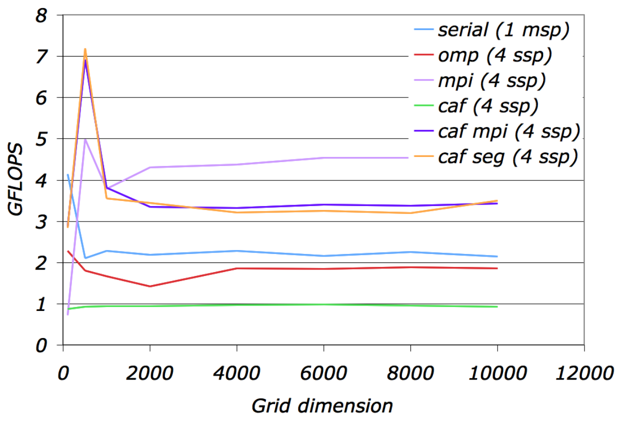
Figures 10(a) and 10(b) show performance within an X1E processor (MSP) in the following modes:

- serial, i.e., compiler-managed parallelism,
- OpenMP, i.e., explicit compiler directive managed parallelism,

[3]Verified by the compiler-generated loopmark listing files as well as the Cray Performance Analysis Tools hardware performance counter reports.



(a) X1E MSP 5-point stencil



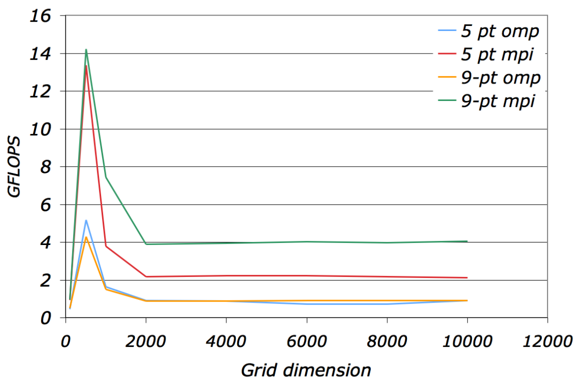
(b) X1E MSP 9-point stenci

Figure 10: X1E Single MSP performance

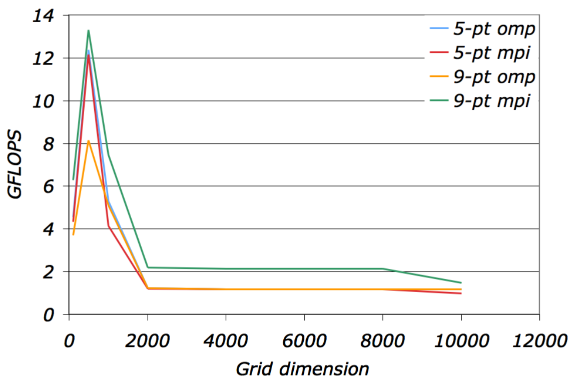
- MPI, i.e., message passing between Fortran processes running on each SSP,
- and Co-Array Fortran, i.e., shared memory loads and stores among local view Fortran processes on each SSP.

For the 5-point stencil, the compiler-generated parallelism is superior to all other implementations (except for very small local grid sizes). Interestingly, for the 9-point stencil, the SSP MPI version is the clear winner. This helps explain the performance of the multi-MSP experiments, where CAF-Shmem and CAF-Segmented outperform MPI for the 5-point stencil but MPI is the best for the 9-point stencil.

Multi-core micro-processor based parallel performance is shown in Figures 11(a) and 11(b). On the AMD



(a) 8 AMD dual-core Opterons (16 cores)



(b) Intel Clovertown

Figure 11: Multi-core micro-processor parallel performance

system, all code was compiled using an Intel Fortran compiler, with SSE instructions enabled (flag `-fastsse`). On the Clovertown system, all code was compiled using the Intel Fortran compiler with optimization flag `-fast`. Again we see differences between the various implementations and stencils. (For both systems, lower process counts mirror the relative performance of the full process counts. In the interest of space, these results are not included.)

5.3 Discussion

The above experiments illustrate how realized performance of even a relatively simple computation can vary significantly based on architecture, algorithm, problem size, and programming model and language.

Because these stencil computations permeate so many different scientific areas and algorithms, extracting available performance is a critical component of overall performance of many large scale application programs. However, the code developer should not be required to implement stencils in all of these different models and their variants. Instead, a language should enable expression of algorithms at a level of abstraction that does not constrain the compiler and run time system, allowing them to take advantage of architecture capabilities. The Chapel language abstraction allows the compiler and runtime system to choose the most appropriate inter-process communication protocol. (Precedent for this approach is shown in [11].)

On the X1E, the CAF-Segmented and CAF-Shmem 5-point stencils outperform the message passing model. For the 9-point stencil, this is also the case until the message length exceeds some threshold where the message latency is smeared across the actual data transmission cost. On machines without support for remote loads and stores, Chapel may be configured using the MPI model.

Within a multi-core processor socket, we continue to see MPI-based implementations out perform their OpenMP counterparts. However, we should not presume this trend will continue as the number of processor cores increases on a socket. Furthermore, other programming models may emerge that make more effective use of the multi-core architecture. In fact CAF implementations may appear that are capable of outperforming MPI.

On yet other machines a combination of approaches might be optimal, or an entirely different protocol might be appropriate[4]. It is not hard to envision a compiler that could recognize this situation, and configure the executable for runtime system decision making. A code developer could do this (as we did), but that works against the other requirements for productive computing.

As previously discussed, Chapel provides a flexible means for distributing the data across the parallel processes. Here we examined only the block decomposition; we intend to experiment with other decompositions on other architectures as they become available.

6 S_N transport

The S_N transport is used to model neutron transport in a deterministic manner. Computation is driven as waves "sweep" across the grid as illustrated in Figure 12(a). On a parallel processing architecture, data must travel with the wave from one distributed parallel process to the next. In a message passing implementation, the relatively small amount of computation required during each time step forces synchronization of runtime execution due to the message passing requirements (illustrated in Figure 12(b)).

A locality-aware implementation was configured[4] for use on a cluster of SGI Origin 2000s, which significantly improved performance on that platform. However, a Chapel implementation would enable this, as well as the message passing implementation, as well as any other implementation, with no changes to user code. For another example, a version was created for use on the Thinking Machines CM-200 which reorganized layout of data in memory so that data would be accessed in a manner closer to the requirements of the algorithm[3]. The sweep algorithm is well-suited for execution on a multi-threaded architecture, such as the MTA-2. However, a message passing version would yield a poor performance; one configured using multi-threading capabilities performs well[7].

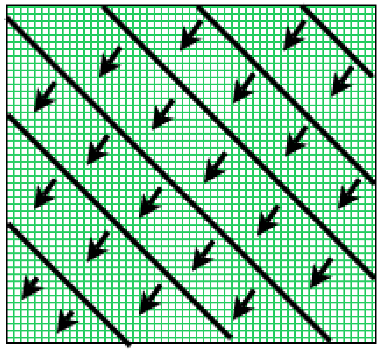
The problem is that each of the above implementations requires distinct coding models. Chapel's global-view combined with its unconstrained memory model could create each of these implementations from the same code.

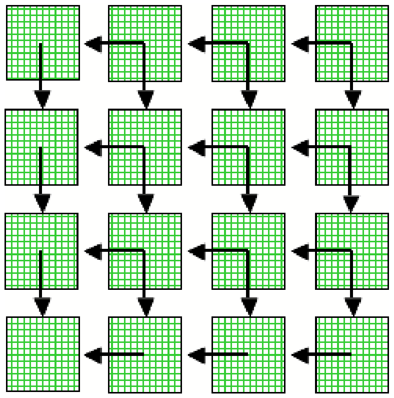
7 Conclusions and Future Work

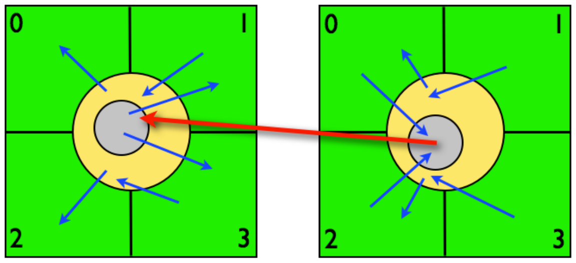
We have demonstrated that the global-view parallel programming model as defined by the Chapel programming language provides a level of abstraction that could allow applications to perform at the highest level achievable by the ubiquitous Fortran-MPI model as well as the Co-Array Fortran and OpenMP models. More importantly, Chapel does so in a manner that gives the compiler the flexibility to exploit architecture-specific runtime system capabilities, resulting in easier-to-write, performance-portable code.

We recognize that in order to achieve this level of performance, the compiler and runtime system may need to take on tasks not normally required of them. And we temper our expectations with the need to properly set expectations: the capabilities discussed in this paper and others will only be possible as the

[4]Current examples include the Cray XMT and the IBM Cell processor. Future architectures promise even greater flexibility.


(a) S_N transport


(b) Message passing



(c) Message aggregation model

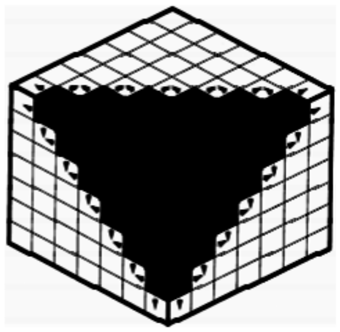

(d) CM-200 sweeping on a 3d grid

Figure 12: S_N transport implementations

compiler technology matures. However, these tasks are relatively well-defined and tractable. For example, selecting the appropriate inter-process communication mechanism is simply a matter of understanding the capabilities of the architecture with regard to the amount of data that must be transmitted between the parallel processes.

We look forward to tracking the progress of the Chapel specification and prototype compiler, both as a means of exploring the expressiveness of the language within the context of important applications and its performance capabilities and potential. We are studying other classes of computations using Chapel, as well as Fortress and X10. More interesting (and more challenging!) are our investigations into how Chapel constructs might influence the development and choice of algorithms for posing computational science experiments.

References

[1] S.R. Alam, R.F. Barrett, J.A. Kuehn, P.C. Roth, and J.S. Vetter. Characterization of Scientific Workloads on Systems with Multi-core Processors. In *IEEE International Symposium on Workload Characterization*, 2006.

[2] 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.

[3] R.S. Baker and K.R. Koch. An S_n Algorithm for the Massively Parallel CM-200 computer. *Nuclear Science and Engineering*, 128, 1997.

[4] R.F. Barrett. Simplifying performance on clusters of shared-memory multi-processor computers. In *BITS: Computing and Communications News*, 2000. `http://library.lanl.gov/cgi-bin/getfile?00393581.pdf`.

[5] R.F. Barrett, S.W. Poole, and S. R. Alam. Expressing POP from a Global View Using Chapel: Towards a More Productive Ocean Model. Technical Report TM-2007/122, Oak Ridge National Laboratory, 2007.

[6] R.F. Barrett, P.C. Roth, and S.W. Poole. Finite Difference Stencils Implemented Using Chapel. Technical Report TM-2007/119, Oak Ridge National Laboratory, 2007.

[7] Larry Carter, John Feo, and Allan Snavely. Performance and programming experience on the tera mta. In *PPSC*, 1999.

[8] 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.

[9] Brad Chamberlain. Chapel development team, Private communication, 2005-07.

[10] P. Charles, C. Donawa, K. Ebcioglu, 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.

[11] Sung-Eun Choi and Lawrence Snyder. Quantifying the effects of communication optimizations. In *IEEE International Conference on Parallel Processing*, 1997.

[12] Cray, Inc. Chapel Language Specification 0.750. `http://chapel.cs.washington.edu`, 2007.

[13] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), 1998.

[14] DARPA. High Productivity Computing Systems program. `http://highproductivity.org`, 1999.

[15] R.E. Diaconescu and H.P. Zima. An Approach to Data Distribution in Chapel. *International Journal on High Performance Computer Applications*, 21(3), 2007.

[16] 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.

[17] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy. The Impact of Multicore on Computational Science Software. *CTWatchQuarterly*, 3(1), February 2007.

[18] 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 Extentions.* The MIT Press, 1998.

[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] Accelerated Stratigic Computing Initiative. The ASCI SWEEP3D Benchmark Code. `http://www.llnl.gov/asci_benchmarks`, 1995.

[21] R.W. Numrich and J.K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998.

[22] Phoenix. Cray X1E at Oak Ridge National Lab. `http://info.nccs.gov/resources/phoenix`.

[23] D.E. Post and L.G. Votta. Computational Science Demands a New Paradigm. *Physics Today*, 58(1), January 2005.

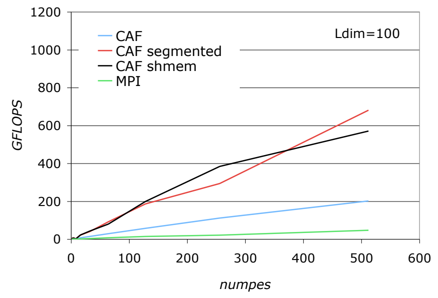
[24] P.C. Roth and J.S. Vetter. Intel Woodcrest: An Evaluation for Scientific Computing. In *8th LCI International Conference on High-Performance Clustered Computing*, 2007.

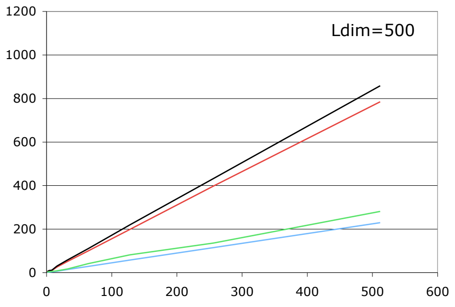
[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference: Volume 1 - 2nd Edition.* The MIT Press, 1998.

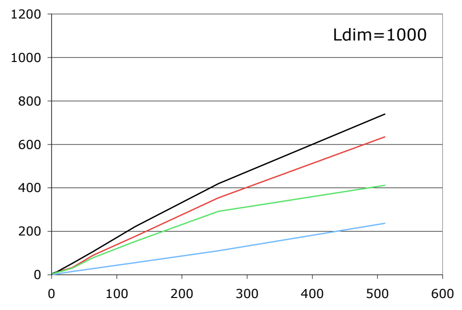
[26] UPC. Consortium, UPC Language Specification. May 31 2005.

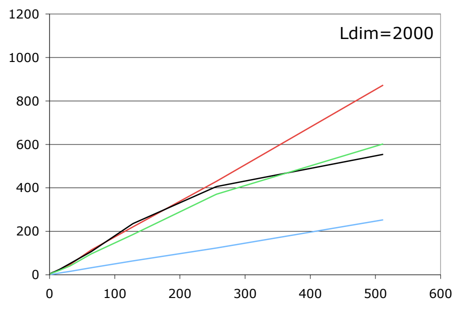
A 5-pt stencil performance on Cray X1E

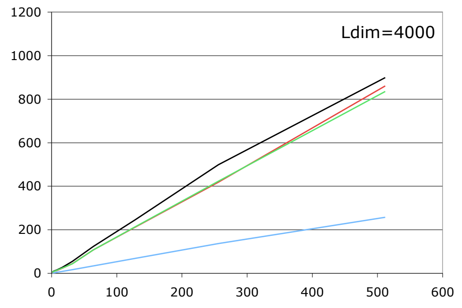
Cray X1E weak scaling performance of the 5-point difference stencil for various MPI and CAF implementations is shown below, for various (local) grid sizes. The x-axis represents the number of MSP processors, the y-axis is GFLOPS. The per processor square grid dimension (`Ldim`) is increasing from the top left graph to the bottom right.

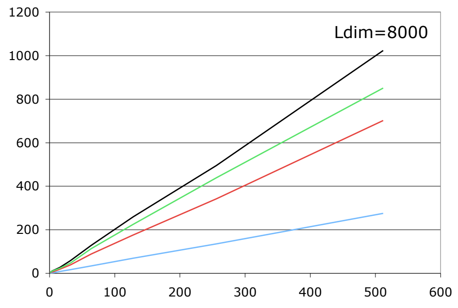






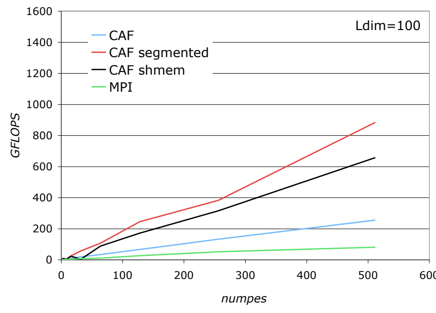






B 9-pt stencil performance on Cray X1E

Cray X1E weak scaling performance of the 9-point difference stencil for various MPI and CAF implementations is shown below, for various (local) grid sizes. The x-axis represents the number of MSP processors, the y-axis is GFLOPS. The per processor square grid dimension (`Ldim`) is increasing from the top left graph to the bottom right.


Ldim=100
CAF
CAF segmented
CAF shmem
MPI
GFLOPS
numpes