Global data re-allocation via communication aggregation in Chapel

Alberto Sanz*, Rafael Asenjo*, Juan López*, Rafael Larrosa*, Angeles Navarro*,
Vassily Litvinov†, Sung-Eun Choi† and Bradford L. Chamberlain†
*Department of Computer Architecture. University of Málaga, Spain.
{asanz,asenjo,juan,rafael,angeles}@ac.uma.es
†Cray Inc., Seattle, WA, USA.
{vass,sungeun,bradc}@cray.com

***Abstract*—Chapel is a parallel programming language designed to improve the productivity and ease of use of conventional and parallel computers. This language currently delivers suboptimal performance when executing codes that perform global data re-allocation operations on distributed memory architectures. This is mainly due to data communication that is done without aggregation (one message for each remote array element). In this work, we analyze Chapel's standard Block and Cyclic distribution modules and optimize the communication routines for array assignments by performing aggregation. Thanks to the expressive power of Chapel, the compiler and runtime have enough information to do communication aggregation without user intervention. The runtime relies on the low-level GASNet networking layer, whose versions of one-sided bulk put/get routines that support strides are particularly useful for us. Experimental results conducted on HECToR (a Cray XE6) and Jaguar (Cray XK6) reveal that the implemented techniques can lead to significant reductions in communication time.**

I. Introduction

It is widely accepted that, nowadays, a serial program is a slow program. But it is also known that the development of a parallel algorithm requires extra work from the programmer, which makes parallel machines difficult to use and unattractive for many traditional programmers. In order to ease the burden on parallel developers, a number of parallel languages (Chapel [1], X10 [2], UPC [3], Charm++ [4], Titanium [5], etc.) have been proposed to improve programmability and performance of current and future parallel machines.

In particular, Chapel is an emerging parallel programming language that is under development and pioneered by Cray Inc. in the context of the DARPA High Productivity Computing Systems (HPCS), an initiative to improve the productivity of parallel programmers. Productivity is defined as a combination of performance, programmability, portability and robustness. Chapel, as well as X10 [2], UPC [3] , Co-Array Fortran [6], and Titanium [5], rely on the *Partitioned Global Address Space* (PGAS) memory model, which enables the user to reason about locality without dealing with the low-level data movement chores. In a distributed memory architecture, the user must express parallelism, deal with data distribution, synchronization and communication. Chapel attempts to develop features that ease the burden of parallel programming by providing abstractions for these tasks and optimizing for common cases.

In a previous study [7] we analyzed the performance of a Chapel version of the Parallel Cyclic Reduction algorithm, or PARACR, for solving tridiagonal systems on distributed memory architectures. PARACR performance was suboptimal, mainly due to a global re-localization of data — in particular, a Block-to-Cyclic data redistribution in which communications were performed element-by-element due to the current Chapel runtime implementation. If we advocate for the success of a High Productivity Parallel Language such as Chapel, then performance is an important issue, and the lack of aggregated communication is a major concern that must be addressed.

This motivates the present work, in which we explore how the compiler and runtime in Chapel can perform aggregation of communication by utilizing low-level bulk communication, thereby reducing communication cost. In particular, we exploit the one-sided strided put/get communication routines of the GASNet [8] low-level networking layer, which is the *de facto* standard communication layer for PGAS languages. GASNet is so successful mainly because: i) it is a well engineered SW with a good interface; ii) it is highly portable, and iii) there are tuned implementations for custom HW (like Cray's). GASNet provides the mechanisms needed to efficiently implement synchronization primitives, memory allocation, collective and scatter/gather operations, and bulk communication routines. The latter are key in this work.

Summarizing, the paper makes the following contributions. First, we extend the implementation of Chapel's Block and Cyclic distributions to perform automatic aggregation of communication in certain key cases. Second, we expose the key features of Chapel that enable such an optimization to be carried out without user intervention. Third, we measure and analyze the performance improvements due to aggregated communication on several Chapel benchmarks.

The rest of the paper is organized as follows. First, we introduce main Chapel features, and dive into the details of

The work has been performed under the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures. This work was also supported in part by the following Spanish projects: TIN2010-16144 from Ministerio de Ciencia e Innovación, and P08-TIC-3500 from Junta de Andalucía.

This work is supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001.

Chapel's implementation of data distribution and the underlying GASNet communication routines in Section II. Then, in Section III we outline our data aggregation techniques and discuss the language features that enabled our approach. In Section IV we elaborate on the experimental results collected on HECToR Cray XE6 and Jaguar XK6 for three different benchmarks. Finally, we discuss related work and conclude.

II. Background

A. General Chapel background

Chapel is a parallel programming language that supports generic OO programming, iterator functions, and type inference. Chapel provides global-view data structures, as well as global-view control flow. An individual processor or node of the target architecture is represented as a *locale* for the purpose of reasoning about locality. Accesses of a task to data are called *local* if the task and data are mapped to the same locale, or *remote* otherwise. Typically, a shared memory architecture is considered a single locale, while a distributed memory multicomputer has as many locales as multicore nodes. The user can specify the number of locales, `numLocales`, on which a Chapel program will run by using the "`-nl <#>`" executable command line flag. There is a built-in array, `Locales`, that represents the set of locales on which the program is running.

The key component for data parallelism in Chapel is the concept of a *domain* [9], a language construct that describes an index space. In this paper we focus on dense rectangular arrays, which provide similar capabilities to arrays in Fortran90, although Chapel also supports associative and sparse arrays. Rectangular domains are a first-class ordered set of Cartesian indices that can have any arbitrary rank and can be iterated over by loops. In addition, domains are used to describe the size and shape of arrays. Summarizing, domains specify index sets and are used to declare, slice and resize arrays. Dense rectangular domains can be represented compactly, using just the bounds, stride and alignment values, per domain dimension.

Data distributions (or *domain maps*) are another first-class concept in Chapel: they can be named, manipulated and passed through functions. In particular, Chapel offers the `dmapped` keyword that allows mapping a domain's indices to the target architecture using the specified distribution. In other words, a distribution is a recipe that Chapel uses to map data (and its associated computation) to the locales where the program executes [10]. Chapel provides a set of commonly-used standard distributions such as *Block*, *Cyclic*, *Block-Cyclic* and *Replicated*. Additionally, it offers support for user-defined distributions via its DSI (Domain map Standard Interface) [9]. These user-defined distributions are developed directly in Chapel and can use all its features (classes, locales, iterators, etc.), which simplifies the programmer's work. A domain map can be a *layout* if it targets a single locale, or a *distribution* if it targets potentially multiple locales. A layout determines how to store and iterate over domains and arrays, whereas a distribution also decides how to map domain indices to locales. An interesting result of this organization is that arrays declared over domains that have been distributed using a single domain map instance, are aligned and therefore locality can be better exploited.

```
1  use BlockDist;
2  config const n=500;
3  var DA = [1..n,1..n,1..n];
4  var DB = [1..2*n,1..2*n,1..2*n];
5  var Dom1 = DA dmapped Block(DA);
6  var Dom2 = DB dmapped Block(DB);
7  var A:[Dom1] real(64);
8  var B:[Dom2] real(64);

10 var D1 = [1..n by 4,1..n by 3,1..n];
11 var D2 = [1..n,1..n by 4,1..n];

13 A[D1] = B[D1]; // First assignment
14 A[D2] = B[D2]; // Second assignment
```

Fig. 1. Chapel code for running example

In Fig. 1 we show Chapel code that will serve as a running example. On line 3 we define `DA` as a $n \times n \times n$ 3D domain that is initialized to contain the set of indices (i, j, k) with $i \in \{1, .., n\}$ and similarly for j and k. Then, on line 5 we declare a domain `Dom1` that has the same indices as `DA` but is mapped to the target architecture using Chapel's standard `Block` distribution. By default the standard distributions map a domain's indices to all locales, arranging the locales into a mesh with as equal dimensions as possible. On line 7 we declare an array `A` over the indices of `Dom1`. Its array elements are 64-bit `real` numbers, mapped to the locales in the same way as the corresponding indices of `Dom1`. We declare `DB`, `Dom2` and `B` in a similar way, but with twice as many indices or elements in each dimension.

Since the array subscripts in Fig. 1 lines 13 and 14 are domains, they result in array *slices*. A slice aliases those elements of the original array whose indices are also in the subscript domain. For n=500, the first assignment results in a data movement (some of them local) of 125 planes of 166 $\times$ 500 elements each. Similarly, the second assignment moves 500 planes of 125 $\times$ 500 elements.

Chapel has built-in support for data parallel computations. The main construct for data parallelism in Chapel is the `forall` loop. It iterates over the indices of a domain or the elements of an array. For each index, the body of the loop is executed by default on the locale where that index is located. Additionally, some operations can be implicitly executed in parallel. For instance, the assignments in lines 13 and 14 of our example are over arrays (slices) and so are data-parallel assignments, executed via implicit forall loops.

Chapel's `on` clause controls where (on which locale) the computations are carried out. For example, this code snippet shows how each locale can create a private copy of an element of array `B`, in parallel:

```
1 forall loc in Locales do on loc {
2   var x:real; //Private on each locale
3   x=B[1, 1, 1]; //Each locale reads B[1, 1, 1]
4 }
```

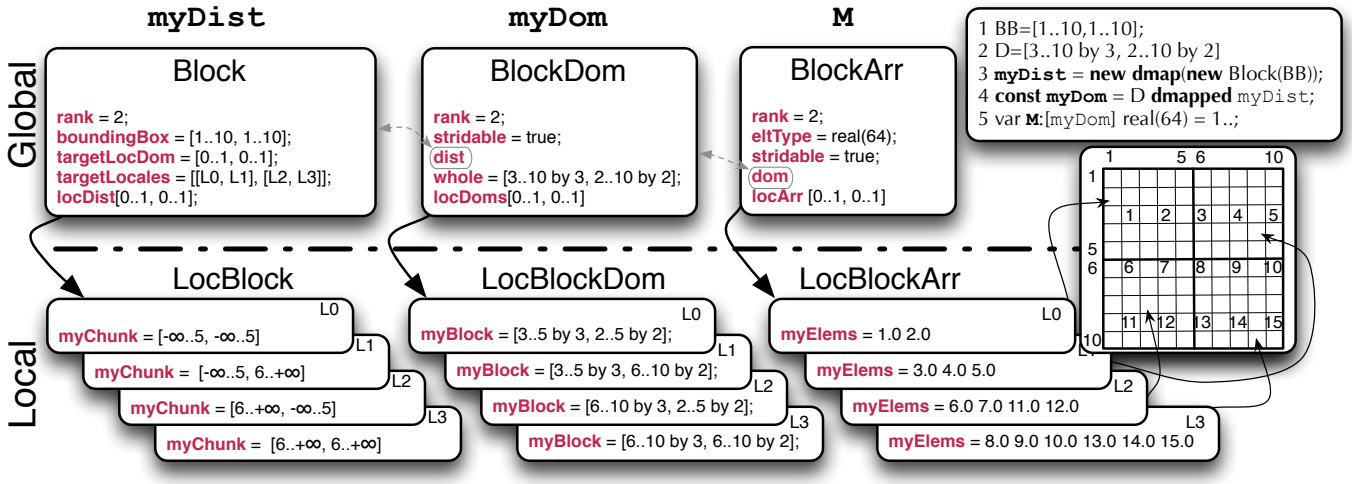


Fig. 2. Illustrating the Block distribution for an array M distributed between 4 locales.

In Chapel, the compiler and runtime systems handle the appropriate communication calls when operations over remote elements must be performed by a task. However, the current implementation of Chapel generates a single point-to-point communication call each time that a remote access is performed. Obviously, in the case of a global re-distribution or partial re-localization of data, aggregation of communications served by bulk communication routines is necessary if we want to improve performance. This is what we describe in more detail next.

B. Data distribution details

The previous section introduced arrays, domains and domain maps, which are the key concepts related to data distribution and data parallelism. Now, we illustrate Chapel's framework for implementing domain maps with the standard Block distribution as an example. The Chapel module `BlockDist` defines three *global descriptor* classes for (i) the distribution or domain map, (ii) a domain, mapped using that domain map, and (iii) an array over that domain. This module is used in the code shown in Fig. 2, which we assume is executed on 4 locales. The code declares the domain map `myDist`, the domain `myDom` and the array `M`. At run-time they are represented using global descriptors, which are instances of the respective descriptor classes `Block`, `BlockDom` and `BlockArr`. There are also *local descriptors*, which are instances of the classes `LocBlock`, `LocBlockDom` and `LocBlockArr`. They are allocated on each locale and store the state corresponding to that locale's subset of the problem space or that is frequently accessed from that locale.

The Block distribution object `myDist` stores the global distribution state: the `boundingBox` field determines the decomposition across locales; `targetLocDom` is the domain that describes the mesh of locales that will host the distributed data; the array `targetLocales[]` identifies the actual locales; and the array `locDist[]` points to the local `LocBlock` descriptors. These per-locale objects store the local distribution boundaries in their `myChunk` fields. Similarly, for the distributed two-dimensional strided domain, `myDom`, the global object `myDom` keeps the overall index set in the field `whole=[3..10 by 3, 2..10 by 2]` and pointers to the four local `LocBlockDom` descriptors. Each of them, in turn, stores the local index set, `myBlock`, computed by intersecting the global domain `myDom.whole` with the corresponding local `myChunk`. It is allowed to have indices outside the `boundingBox` grid. For example, if `myDom` were `[-5..5, -2..4]`, it would be mapped completely to locale 0. Finally, array `M` is declared over `myDom` and initialized with the sequence `1..` (i.e. 1, 2, 3, 4, ...). The global array descriptor, a `BlockArr` instance, identifies the element data type (64-bit real number) and points to its global domain descriptor. The actual elements of the array `M` are stored locally in each local descriptor's array `myElems`. The `myElems` arrays are mapped using the default layout, `DefaultRectangular`, so will be called *DR* arrays in this discussion. Among Chapel arrays, DR arrays represent physical memory most directly.

C. GASNet support

For the sake of portability, Chapel relies on GASNet [8] as the communication layer, a network-independent and language-independent high-performance communication interface intended for use in implementing the runtime system for Partitioned Global Address Space languages. The GASNet Extended API provides blocking and non-blocking memory-to-memory transfer operations for contiguous and strided bulk data. As an example, the operation `gasnet_get(void *dest, gasnet_node_t node, void *src, size_t nbytes)` fetches `nbytes` bytes from the address `src` on node `node` and places them at `dest` in the local memory space.

In addition, GASNet provides non-blocking, split-phase memory accesses to shared data. All such non-blocking operations require an initiation (generally a `put` or `get`) and a subsequent synchronization on the completion of that

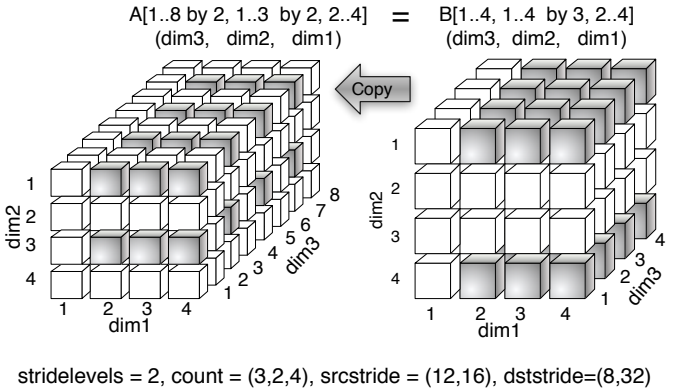


Fig. 3. A=B data transfer example

operation before the result is guaranteed. In particular, for strided memory copy transfers, GASNet interface provides `gasnet_puts_bulk()` and `gasnet_gets_bulk()` as well as the corresponding non blocking (nb) counterparts. These functions have the following arguments: i) `stridelevels` is the number of different strides involved in the transfer (see [8]); ii) a `count` array indicates the slice size in each dimension (`stridelevels+1` entries, where `count[0]` is the number of bytes of contiguous data (chunk) in the leading (rightmost) dimension); iii) `srcaddr/dstaddr` are the starting addresses of the source/destination memory; and iv) the `srcstrides/dststrides` arrays contain the source/destination strides in each dimension.

The strides in `srcstrides` and `dststrides` must be monotonically increasing and must not specify overlapping locations. We have chosen these GASNet operations because they are general enough to allow an efficient implementation of Chapel array slice assignments.

Based on these operations we have implemented two new Chapel primitives, `chpl_comm_gets` and `chpl_comm_puts`, that have the same arguments and essentially convert Chapel DR arrays to the C arrays that match the GASNet interface.

As an example, Fig. 3 shows the communication involved in the Chapel assignment `A[1..8 by 2, 1..3 by 2, 2..4] = B[1..4, 1..4 by 3, 2..4]`, where A and B are DR arrays. In that example, `dstaddr` should point to element `A[1,1,2]` and `srcaddr` to `B[1,1,2]`, while the value of the other arguments involved in the bulk function call are indicated in the figure. In Chapel, the assignment is valid if the number of elements per dimension for both specified domains is the same.

III. Data aggregation implementation

In this work, we focused on reducing the number of data transfers for assignments between Block-distributed arrays and between a Block-distributed array and a Cyclic-distributed array. Taking advantage of Chapel's OOP facilities, we provided specialized implementions of assignment by overloading the = function.

Since a Block-distributed, or *BD*, array is implemented via local DR arrays, we first implemented bulk communication for assignment between two DR arrays. On top of that, we implemented the assignment between a DR array and a BD array, and then the assignment between two BD arrays. We illustrate this organization in Fig. 4 with four locales.

Fig. 4(a) shows the assignment `A[D1]=B[D2]` where `A` and `B` are DR arrays, `A` allocated on locale 0 and `B` on locale 1. Here the runtime needs to compute the arguments for and execute just one call to `chpl_comm_gets` (if the statement is executed on locale 0) or `chpl_comm_puts` (if the statement is executed on locale 1), as described in subsection II-C.

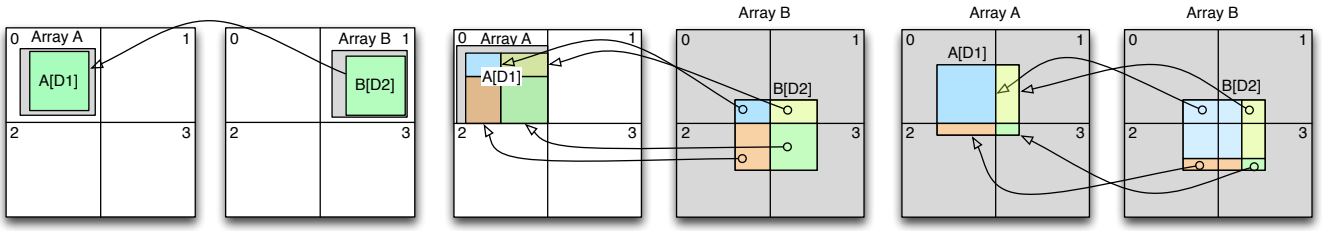
If `B` is a BD array, its elements are stored in several DR arrays, the `myElems` fields of B's local `LocBlockArr` descriptors. If `A` is a DR array, `A[D1]=B[D2]` translates into several DR=DR operations. For example, in Fig. 4(b), B[D2] is spread over 4 locales, whereas A is still stored on locale 0. Here, we divide A[D1] into four regions, $A[D1_0]$ through $A[D1_3]$, such that $A[D1_i]$ is the destination of those elements of B[D2] that are stored on locale i, with $i \in 0..3$. Then we execute four DR=DR assignments in parallel: $A[D1_i]$=`B.locArr[i].myElems[src]`, $i \in 0..3$. For each i, `src` represents the slice of `B.locArr[i].myElems` that corresponds to B[D2] on locale i. The puts or gets bulk function for the assignment on locale 0 implements it internally with a `memcpy`.

Finally, if `A` and `B` are both Block-distributed arrays, we split the problem into several DR=BD cases, since the destination BD array is just a set of DR ones. In Fig. 4(c), the array `A` is stored in four local DR arrays. Therefore the assignment `A[D1]=B[D2]` translates into:

```
1 forall i in 0..3 do
2   on A.locArr[i] do //co-locate with locArr[i]
3    A.locArr[i].myElems[dest] = B[src] //DR=BD
```

where `src` and `dest` are the source and destination domain slices that indicate which portions of the arrays are to be assigned on each locale. In this example `A.locArr[0].myElems` needs 4 regions from `B`, which are stored on four locales. `A.locArr[1].myElems` only needs information from locales 1 and 3. The assignment to `A.locArr[3].myElems` turns out to be a `memcpy` in this example.

We optimize data movement upon assignments from Block-distributed to Cyclic-distributed and from Cyclic-distributed to Block-distributed arrays in a similar manner, converting them to calls to `chpl_comm_gets/puts`. All these data redistributions are special cases of copying one n-dimensional rectangular prism region to another with a different shape, as described in section II-C. For example, for a Block to Cyclic redistribution, each source locale determines the chunk of data to be sent to each destination locale. For each chunk and for each destination locale, the source locale calls `chpl_comm_puts` with the appropriate arguments. In general, this leads to all-to-all communication. The Cyclic-to-Block case is similar, with the destination locales invoking `chpl_comm_gets` instead.



(a) DR=DR (b) DR=BD (c) BD=BD

Fig. 4. A[D1]=B[D2] when A and/or B are DR (Default Rectangular) or BD (Block Distributed) arrays.

```
1  config const n=500;
2  var D1 = [1..n,1..n];
3  var D2 = [1..2*n,1..2*n];
4  var A: [D1] dmapped Block(D1) real;
5  var B: [D2] dmapped Block(D2) real;

7  var DA = [101..200 by 2, 51..200 by 3];
8  var DB = [201..700 by 10, 301..600 by 6];
9  A[DA] = B[DB];
```

Fig. 5. Chapel code to illustrate the mapping functions

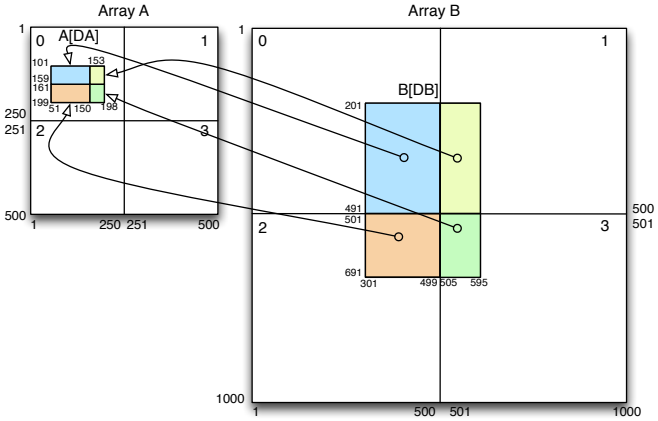


Fig. 6. Illustrating the assignment A[DA]=B[DB] of Fig. 5 when using 4 locales.

A. Subregion computation

As we have seen in the previous section, during the array slice assignment, different subregions of the slice have to be computed, as $D1_i$ and `src` subregions. In order to do that in the case of the assignment `A[DA]=B[DB]`, we first need a correspondence or mapping from destination domain, `DA`, to the source domain, `DB`. This is formally described as follows. Let:

`DA=[`$la_1..ha_1$ `by` sa_1, ...,$la_n..ha_n$ `by` sa_n`]`
`DB=[`$lb_1..hb_1$ `by` sb_1, ...,$lb_n..hb_n$ `by` sb_n`]`

where `n` is the number of dimensions (rank) of the domains `DA` and `DB`. A vector with all the strides, sa_i, can be obtained by `A._value.dom.whole.stride` and similarly for B. Let `[`a_1, a_2,...,a_n`]` and `[`b_1, b_2,...,b_n`]` be some coordinates of the domains `A` and `B` respectively. We define the bijective function `m:DB`$\rightarrow$`DA` so that `(`a_1,a_2,...,a_n`) = m(`b_1,b_2,...,b_n`)` where:

$$a_i = la_i + sa_i \times (b_i - lb_i)/sb_i,\ i \in 1..n$$

The corresponding inverse function $\texttt{m}^{-1}$`:DA`$\rightarrow$`DB`, results in `(`b_1,b_2,...,b_n`)` $= \texttt{m}^{-1}$`(`a_1,a_2,...,a_n`)` where:

$$b_i = lb_i + sb_i \times (a_i - la_i)/sa_i,\ i \in 1..n$$

In order to illustrate how these mapping functions, m and m^{-1}, are used, let's rely on a 2D simplified version of the running example that we can see in Fig. 5. If this code is executed on four locales, Fig. 6 illustrates the distribution and subregions that have to be moved, were we can see that `A[DA]` happens to be on locale 0 while `B[DB]` is distributed between the 4 locales.

When the assignment `A[DA]=B[DB]` is executed, two new alias arrays are created, A'=A[DA] and B'=B[DB]. Now, for this DR=BD case, as we said, we have to execute this loop:

```
1  forall i in 0..3 do
2    on A' do //co-locate with A'
3      A'[Di] = B'.locArr[i].myElems //DR=DR
```

So the problem to solve now is the computation of subregions D_i, $i \in 0..3$. This is done using the mapping function m. For example, for i=0, the subdomain of B stored on locale 0 is given by B'.dom.locDoms[0].myBlock = [201..491 by 10, 301..499 by 6]. We then obtain D_0 using function m to calculate the corresponding ending coordinates of the destination region, $[lxa_1..hxa_1$ by $2, lxa_2..hxa_2$ by 3]:

$$\begin{aligned} lxa_1 &= 101 + 2 \times (201 - 201)/10 = 101 \\ hxa_1 &= 101 + 2 \times (491 - 201)/10 = 159 \\ lxa_2 &= 51 + 3 \times (301 - 301)/6 = 51 \\ hxa_2 &= 51 + 3 \times (499 - 301)/6 = 150 \end{aligned}$$

So we get D_0=[101..159 by 2, 51..150 by 3]. And similarly,

- D_1= [101..159 by 2, 153..198 by 3]
  = M(201..491 by 10, 505..595 by 6)
- D_2= [161..199 by 2, 51..150 by 3]
  = M(501..691 by 10, 301..499 by 6)
- D_3= [161..199 by 2, 153..198 by 3]
  = M(501..691 by 10, 505..595 by 6)

where M converts the low and high bounds of its range arguments using the function m.

As we said, if `A` and `B` are both Block-distributed arrays, we split the problem into several DR=BD cases as we showed in Fig. 4(c), but now we need the m^{-1} function. So, if

the assignment operation is `A[DA]=B[DB]`, two new aliases are created as before: A'=A[DA] and B'=B[DB], and the following code has to be executed:

```
1 forall i in 0..3 do
2   on A'.locArr[i] do //co-locate with locArr[i]
3    A'.locArr[i].myElems = B'[D_i] //DR=BD
```

so we have to find subregions D_i, $i \in 0..3$. But now we use m^{-1} function in order to compute the D_i subdomain of B that correspond to each A'.dom.locDoms[i].myBlock local domain. For example, for the previous example but with DA=[101..400 by 2, 51..350 by 3] and DB=[201..500 by 2, 151..450 by 3] and 4 locales, we get A'.dom.locDoms[0].myBlock=[101..250 by 2, 51..250 by 3] and D_0=[201..350 by 2, 150..350 by 3]. Similarly for D_1 and so on.

For further implementation details the reader is referred to the source code, which is available at:

https://chapel.svn.sourceforge.net/svnroot/chapel/branches/collaborations/bulkComms/modules/{dist/BlockDist.chpl,internal/DefaultRectangular.chpl}

See in particular the functions `doiBulkTransferStride` in each file.

B. Discussion: enabling language features

One of Chapel's most notable design decisions was to allow domain maps to be implemented in Chapel itself. An example of that, the Block distribution, was shown in subsection II-B. This makes Chapel's OOP and productivity features available to the domain map implementor, alleviating the burden of such complex work. In addition, users interested in performance tuning have access to the details of the distribution implementation and can implement further optimizations. Data aggregation in this work is an example.

The cornerstone features that most contributed to easing our task are the global and per-locale data structures that describe the domains of distributed arrays, the underlying support for bulk strided communication in GASNet, and the OOP implementation of domain maps that enables modular construction of our new data transfer methods. Most other PGAS languages do not provide these features; for example, UPC relies on manually called collective routines to reduce communication overhead. On the other hand, X10, which supports domains (called regions) and OO-implemented data distributions, can benefit from the techniques presented in this paper.

IV. Experimental results

Several experiments have been conducted on two Cray supercomputers: HECToR and Jaguar. HECToR is a Cray XE6, Opteron 6276 16C 2.30 GHz with a Cray Gemini interconnection network. This machine is composed of 704 compute blades, each blade contains four compute nodes (locales), each one with two 16-core AMD Opteron 2.3 GHz Interlagos processors. Jaguar is a Cray XK6 that is comprised of 18,688 nodes, each node containing a 16 core AMD Opteron 6274 2.2 GHz processors with Gemini 3D torus interconnect. Jaguar is positioned number 6 (peak capacity of over 2.6 Pflop/s) and HECToR number 32 (over 800 Tflop/s) in the latest top500.org list (Jun. 2012).

In both machines, Chapel has been compiled in order to take advantage of the Gemini conduit available in the underlying GASNet library, but experiments with the default MPI conduit were also conducted in order to compare them, and also because the Gemini conduit is still a beta version. We set variables `GASNET_VIS_AMPIPE=1` and `GASNET_VIS_REMOTECONTIG=1`, in order to enable respectively, packing of most non-contiguous puts/gets and RDMA optimization of locally non-contiguous but remotely contiguous transfers. All Chapel codes where compiled with the `--fast` flag to enable all compiler optimizations.

A. Running example array assignment

First, we compared the Block distributed array assignment when using the default array copy implementation available in the latest release of Chapel (version 1.5) and then relying on the aggregation communication support described in the previous section. We measured the assignment time for the two 3D array assignments of lines 13 and 14 shown in the code snippet of Fig. 1. We will refer to them as the D1 and D2 assignments.

In Fig. 7(a) and (b) we compare the times in seconds for D1 and D2 when n=500, for 2 to 16 locales, using GASNet MPI and Gemini conduits, with and without aggregation of communication, on HECToR and Jaguar. Fig. 7(c) compares the times with aggregation between the MPI and Gemini conduits on HECToR and Jaguar.

In general, times are higher for the D2 assignment because a greater number of elements are involved. With the MPI conduit (Figs. 7(a)), we can see that on HECtoR the aggregation reduces the times by more than three orders of magnitude, whereas on Jaguar the impact is much smaller (around a 30% improvement in some cases). This suggests a better tuned configuration of MPI on Jaguar. On the other hand, the Gemini conduit (Figs. 7(b)) shows a different behaviour: both on HECToR and Jaguar, aggregation reduces the time considerably, but now the impact of aggregation is more significant on Jaguar (more than two orders of magnitude) than on HECToR (less than 10x).

Another interesting result is that the Gemini conduit exhibits a better scalability behavior, surprisingly very good on both machines when no aggregation is performed. This is due in part to the reduction of the number of injected messages per locale when the number of locales increases. That number of injected messages may be a serial bottleneck on a small number of locales.

Figs. 7(c) allows us to focus on the performance of our aggregated implementation, where only one bulk message is injected per locale. We find smaller times on Jaguar (both for MPI and Gemini) than on HECToR, possibly because latencies of one-sided bulk communications on Jaguar are smaller. We also see that Gemini times are always smaller than MPI ones on each machine, especially when the number of locales increases. We think this is due to better communication

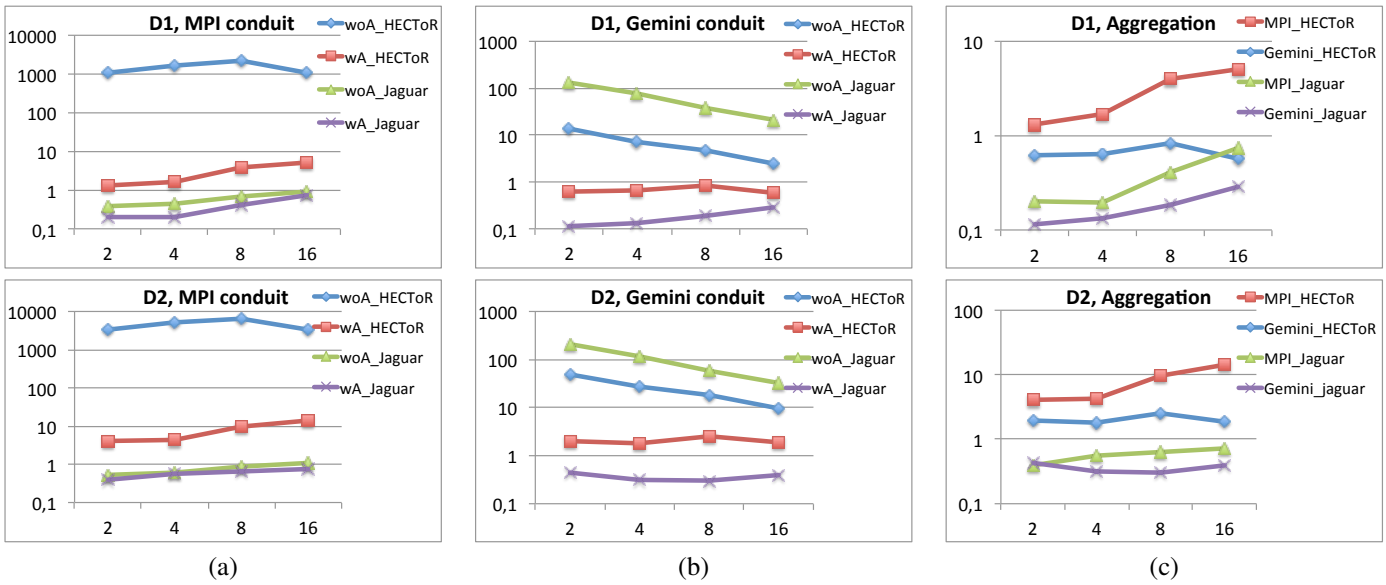


Fig. 7. Time in seconds for array assignmet A=B on different numbers of locales, without/with aggregation (woA/wA) under: a) MPI conduit, b) Gemini conduit and c) MPI-Gemini comparison.

scheduling by the one-sided bulk functions in the Gemini conduit.

The next subsections evaluate our implementation on two numerical codes, FFT and PARACR (a tridiagonal solver). We focus on the Gemini conduit, since it has proven to outperform the MPI one.

B. The radix-4 FFT algorithm

Our evaluation used a Chapel implementation of the FFT algorithm that is part of the HPC Challenge (HPCC) benchmark suite [11]. It uses radix-4 butterflies and is divided into two main phases, the first one using a Block distribution and the second one a Cyclic distribution. When run on 4^k locales, this guarantees that butterfly-patterned accesses are completely local. Communication occurs only between phases, when copying the vector from Block to Cyclic (BtoC) and from Cyclic to Block (CtoB).

Table 8 shows the times in seconds of the FFT code on HECToR and Jaguar, for 1 to 16 locales, when vector size is $n = 2^{20}$ and the Gemini conduit is used. We indicate the total execution time ("T. total"), and the BtoC and CtoB redistribution times. Again, we compare the cases of no aggregation vs. aggregation of communications. The column "Comm. Imp." lists the ratio of the total redistribution time (BtoC plus CtoB) between the two cases, showing a significant improvement in the aggregated version. In the single-locale case we see the impact of reducing the number of calls to memcpy.

From the table we see that BtoC is usually faster than CtoB, especially in the non-aggregated case. This is because in our setup, BtoC and CtoB execute on #cores pthreads per locale. The current implementation of the second phase of the FFT has a nesting of two forall loops that creates $(\#cores)^2$ pthreads after BtoC but before CtoB. CtoB is assigned to an arbitrary #cores-subset of them, which tend to be mapped to, and compete for, a subset of cores while others idle. In any case, the aggregation of communications has a significant impact on both machines. On Jaguar in particular we see a $80\times$ improvement. In the aggregated case, for both BtoC and CtoB, we benefit from the low-level RDMA optimization enabled by `GASNET_VIS_REMOTECONTIG=1`: locally non-contiguous data that arrive from/are sent to a remote contiguous memory region are packed more efficiently.

C. The PARACR algorithm

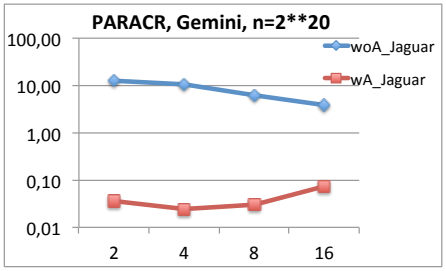
Parallel Cyclic Reduction (called PARACR in [12]) is a well known algorithm for solving tridiagonal systems of equations. It proceeds in two phases: Substitution and Solution. For a problem of size n, the Substitution phase consists of $O(log\ n)$ steps and at each step $O(n)$ butterflies are computed.

In a previous study [7] we analyzed the performance of a Chapel PARACR implementation and observed that redistributing data from Block to Cyclic (BtoC) at a certain intermediate step improves performance. This is because the Block distribution favors data locality during early steps, while the Cyclic distribution takes advantage of locality at later steps. Now we can carry out the BtoC redistribution taking advantage of aggregated communications.

Fig. 8 shows the times of the BtoC redistribution in the PARACR code on Jaguar under the Gemini conduit for 1 to 16 locales. Note that in this code four arrays —each one of size $n = 2^{20}$— have to be redistributed. Therefore, the time for the redistribution when there is no aggregation is around 4 times higher than for FFT (where only one array is redistributed). Aggregation again brings a significant speedup, from two to three orders of magnitude. As in the other experiments under the Gemini conduit, the aggregated communications case shows a slight increase in time when the

FFT on HECToR, Gemini conduit, n=2^{20}							
	Without aggregation			With aggregation			
Loc	T. Total	T. BtoC	T. CtoB	T. Total	T. BtoC	T. CtoB	Comm. Imp.
1	355.485	0.850	6.6653	351.180	0.0109	0.0099	361.31
4	82.162	0.944	5.4038	75.937	0.3032	1.3381	4.63
16	71.963	2.143	5.1227	65.476	0.2454	0.4146	11.00
FFT on Jaguar, Gemini conduit, n=2^{20}							
	Without aggregation			With aggregation			
Loc	T. Total	T. BtoC	T. CtoB	T. Total	T. BtoC	T. CtoB	Comm. Imp.
1	371.52	0.0175	0.0761	375.07	0.00495	0.03987	2.09
4	224.92	3.1547	4.6204	222.66	0.01095	0.08519	80.88
16	131.93	1.3238	1.5435	123.28	0.03310	0.05919	31.07

(a)



(b)

Fig. 8. a) times in seconds for FFT on HECToR and Jaguar; b) Jaguar times in seconds for BtoC redistribution in PARACR.

number of locales increases, due to the fact that the number of bulk messages per locale is $O(\#locales)$. However, in the non-aggregated case the time tends to decrease. Indeed, the total number of point-to-point messages equals the number of array elements n, so the number of messages per locale is $O(n/\#locales)$.

V. Related work

There have been several efforts over years to improve productivity of large-scale parallel programming. A significant effort has been devoted to provide highly efficient libraries that run over MPI, such as ScaLAPACK [13], or PBLAS[14] or PETSc[15]. Although these approaches hide the data distribution and communication operations from the user, their applicability is limited to the numerical kernels that they support. By contrast, many language initiatives like ZPL [16], UPC [3], Co-Array Fortran [6], X10 [2], Chapel [1] and Titanium [5] have provided high-level constructs to express data parallelism easily, striving for better programmability and generality while achieving performance. Among them, the Partitioned Global Address Space (PGAS) languages offer programmers a shared address space model that simplifies programming while exposing data/thread locality to enhance scalability. Chapel and UPC support the concept of global view arrays, which means that an array is declared and operated upon as if it were a single logical array even though it is distributed among multiple distributed-memory nodes. On the contrary, arrays in Co-Array Fortran and Titanium are not globally viewed since users work with distributed arrays as local-per nodes arrays. One distinguishing feature of Chapel is that, in addition to supporting multidimensional standard Block and Cyclic distributions, it also provides a general framework for defining user-defined distributions.

PGAS languages are designed around a data-centric memory model in which the remote accesses are performed through one-sided (put/get) communication libraries (typically GASNet). One open question is how these languages will manage global data re-localization operations to improve performance. Generally, this is performed by using collective communication operations. For instance, one proposed extension to UPC [17] incorporates a new interface to express two types of collectives: one-to-many (e.g. broadcast) and many-to-many (e.g. exchange). The user has to specify the blocks of data involved in the source and destination arrays and let the underlying runtime system handle the problem of mapping data blocks to tasks and packing/unpacking of data. Another proposed extension to collectives in Co-Array Fortran [18] advocates expressing collective communication using co-array section notation and get semantics. However, Co-Array Fortran shared arrays are limited to Block distributions, which reduces their applicability. Our approach in Chapel is to hide the details of communication management from the user. When a data re-localization operation is to be performed, we rely on the runtime to identify which tasks participate, to infer what type of bulk operation should be performed, to gather the appropriate information about how to pack the data, and to call the corresponding one-sided stride communication GASNet routine.

In the context of PGAS languages, recent studies addressed the interaction of the communication layer with the scheduling layer [19]. Other studies focused on communication/computation overlap using one-sided communication routines provided by PGAS [20] as well as compilation techniques to coalesce remote accesses into a single one [21]. We consider these studies complementary to our work.

Recent work in X10 focussed on compiler optimizations and transformations to reduce communication overheads [22]. In particular, they combined scalar replacement with transformations such as loop distribution, scalar expansion, loop tiling and loop splitting, to reduce the size of the communication buffer and to avoid redundant communication data. These redundant data are related to the reference handles for the objects that are dispatched to remote locales, or whole arrays that are sent to remote locales when a reference to any element of such an array is performed on a remote locale. However, they did not address bulk communications to handle the problem of global data re-localization operations, which is the topic of this paper.

VI. Conclusions

Chapel is an emerging parallel programming language designed to facilitate programming of current –and future– generation parallel systems. However, one important issue is how the language can provide application scalability while ensuring programmer productivity. In this context, our paper explores how the compiler and runtime systems can

perform aggregation of communications when global data re-localization operations are carried out. We studied two standard Chapel distributions, Block and Cyclic, and optimized their implementations to reduce communication costs. We took advantage of GASNet one-sided bulk communication routines and delegated to the runtime the burden of identifying what regions of data need to be moved to which locales. Then the minimum number of communication invocations is performed. Our experimental results show that a significant reduction in communication times is achieved, especially when GASNet's default MPI conduit is used. In the future we plan to improve the scheduling of bulk communications to ensure that our re-localization operations scale to a large number of locales.

ACKNOWLEDGMENTS

This work made use of the facilities of HECToR, the UKs national high-performance computing service, which is provided by UoE HPCx Ltd at the University of Edinburgh, Cray Inc and NAG Ltd., and funded by the Office of Science and Technology through EPSRCs High End Computing Programme. This research also used resources of the Oak Ridge Leadership Computing Facility located in the Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725. We also thank Patrick Worley who granted us access to Jaguar as part of the CSC023 research project.

REFERENCES

[1] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Intl. J. of High Perf. Computing Applications*, vol. 3, no. 21, pp. 291–312, August 2007.

[2] 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 clustered computing," in *OOPSLA*, 2005.

[3] U. Consortium, "UPC language spec, v1.2," LBNL-59208, Tech. Rep., 2005.

[4] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[5] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick, "Titanium language ref. manual," CS Division (EECS), UC, Berkeley, CA, Tech. Rep., 2001.

[6] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998.

[7] A. Sanz, J. López, A. Navarro, and R. Asenjo, "Tridiagonal systems in Chapel. case study: the parallel cyclic reduction algorithm," in *XXI J. de Paralelismo*, 2010, pp. 59–66.

[8] D. Bonachea, "Proposal for extending the UPC memory copy library functions and supporting extensions to GASNet, Version 2.0," LBNL, Tech. Rep., 2007.

[9] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, D. Iten, and V. Litvinov, "Authoring user-defined domain maps in Chapel," in *CUG'11*, June 2011.

[10] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-defined distributions and layouts in Chapel: Philosophy and framework," in *2nd USENIX Workshop on Hot Topics in Parallelism*, June 2010.

[11] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and D. Iten, "HPC challenge benchmarks in Chapel," 2009, Cray Inc.

[12] R. W. Hockney and C. R. Jessope, *Parallel Comp.: Archit., Programming and Algorithms*. Ed. Adam Hildger, 1988.

[13] L. S. Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker, "Scalapack: a portable linear algebra library for distributed memory computers - design issues and performance," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, 1996.

[14] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. C. Whaley, "A proposal for a set of parallel basic linear algebra subprograms," in *Proceedings of the Second Intl. Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science*, 1996, pp. 107–114.

[15] S. Abhyankar, B. Smith, H. Zhang, and A. Flueck, "Using petsc to develop scalable applications for next-generation power grid," in *Proceedings of the First Intl. Workshop on High Performance Computing, Networking and Analytics for the Power Grid*, ser. HiPCNA-PG '11, 2011, pp. 67–74.

[16] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, "ZPL: A machine independent programming language for parallel computers," *IEEE Trans. Softw. Eng.*, vol. 26, no. 3, pp. 197–211, Mar. 2000.

[17] R. Nishtala, G. Almasi, and C. Cascaval, "Performance without pain = productivity: data layout and collective communication in UPC," in *PPoPP '08*, NY, 2008, pp. 99–110.

[18] M. Sottile, C. Rasmussen, and R. Graham, "Co-Array collectives: refined semantics for CAF," in *ICCS'06*, 2006, pp. 945–952.

[19] F. Blagojević, P. Hargrove, C. Iancu, and K. Yelick, "Hybrid PGAS runtime support for multicore nodes," in *PGAS '10*, 2010, pp. 1–10.

[20] C. Iancu, W. Chen, and K. Yelick, "Performance portable optimizations for loops containing communication operations," in *ICS '08*, NY, USA, 2008, pp. 266–276.

[21] C. M. Barton, "Improving access to shared data in a partitioned global address space programming model," Ph.D. dissertation, Univ. of Alberta, 2009.

[22] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlić, and V. Sarkar, "Communication optimizations for distributed-memory X10 programs," in *IPDPS'11*, 2011.