A Team-Based Methodology of Memory Hierarchy-Aware Runtime Support in Coarray Fortran

Dounia Khaldi*, Deepak Eachempati*, Shiyao Ge* Pierre Jouvelot† and Barbara Chapman*
*Department of Computer Science
University of Houston, Houston, Texas
Email: {dkhaldi, dreachem, sge2, bchapman}@uh.edu
†MINES ParisTech, PSL Research University, France
Email: pierre.jouvelot@mines-paristech.fr

***Abstract*—In this paper, we describe how a 2-level memory hierarchy can be exploited to optimize the implementation of teams in the parallel facet of the upcoming Fortran 2015 standard. Teams have been suggested to leverage the hierarchical parallelism present in many applications. They are supposed to be used as an extension to Coarray Fortran (CAF), the explicitly-parallel part of the Fortran 2008 standard that adheres to the Partitioned Global Address Space (PGAS) programming model. Specifically, we focus on reducing the cost associated with moving data within a node and between nodes, finding that this distinction is of key importance when looking at performance issues. We introduce a new hardware-aware approach for PGAS, to be used within a runtime system, to optimize the communications in the virtual topologies and clusters that are binding different teams together. We have applied and implemented this methodology to three important collective operations, namely barrier, all-to-all reduction and one-to-all broadcast.**

To validate our approach, we implemented full support for teams into the CAF OpenUH compiler. To the best of our knowledge, the resulting platform is the first Fortran compiler that both provides teams and handles such a memory hierarchy methodology within teams. We illustrate the benefits of our methodology on a new Team Microbenchmark suite we specifically developed for this research and also on High Performance Linpack (HPL). Our memory hierarchy-awareness approach for barrier, reduction and broadcast operations shows up to 26-, 74- and 3-fold performance improvements over the default approach, respectively.

***Keywords*-Coarray Fortran; teams; PGAS; memory hierarchy; intra- and inter-node runtime; collective operations**

I. Introduction

The emergence of many-core compute nodes in large-scale computing systems, such as the currently top-ranked Tianhe-2[1], requires programming model implementers to consider more carefully the memory system hierarchy when looking at and overcoming performance issues. Rather than adding low-level language features to exploit the performance gains permitted by such architectures, likely to result in programs that are non-portable and difficult to maintain, the *implementation* of the programming model for a given platform should be responsible for detecting such system characteristics at run time and exploiting this information. As an illustrative example, consider the dissemination barrier [1], a commonly used algorithm for barrier synchronization between P processors commonly employed on distributed memory systems due to its strong scaling properties. When the P processors are distributed over Q nodes, with multiple processors per node and the bandwidth between the nodes being limited, a straightforward execution of this algorithm will yield poor results as it may result in multiple processors contending for the inter-node bandwidth.

Most parallel applications are programmed using the Message Passing Interface (MPI) [2], where multiple processes execute in a coordinated manner, communicating by performing *send* and *receive* operations. More recently, several languages and libraries have added support for explicit or implicit remote memory access (RMA) using so-called "one-sided communication", including languages following the *Partitioned Global Address Space* (PGAS) paradigm as well as MPI (MPI-2 added RMA to the interface and MPI-3 made significant refinements to better support it). Of special note is the Fortran 2008 addition for supporting *coarrays*, a language mechanism that enables RMA as a natural extension to Fortran's array syntax, informally named CAF[2]. In this paradigm, an *image* represents an executing process in an SPMD program with its own copy of data.

The purpose of this paper is to provide new optimization strategies for coarrays. We suggest to decompose applications into subproblems that may be worked upon concurrently, and organize this work among subsets of image *teams*. Additionally, we consider the ensuing challenge of reducing the costs associated with moving data within a node and between nodes. Our approach is thus to combine a hierarchical decomposition of applications across two dimensions: (1) a logical partitioning of the work, based on the application, and (2) a processor layout hierarchy, based

[1]Currently, Tianhe-2 is the world's fastest supercomputer according to the TOP500 list dated November 2014.

[2]This acronym describes the Co-Array Fortran extension proposed by Cray Computer several years before it was adopted into the standard. We refer to the implementation of CAF in the OpenUH compiler as UHCAF. CAF 2.0 is an alternative Fortran language extension for supporting coarrays proposed by Rice University.

on the underlying hardware.

The first dimension we address will use teams for coarray programs. Teams are expected to be adopted in Fortran 2015, and they are described in a draft of the technical specification for additional parallel processing features[3]. This concept has been already introduced in many parallel programming APIs such as MPI [2], in the form of communicators. (We discuss related work in Section VI.) Image teams make it possible to divide applications into loosely coupled subproblems that are handled by different subsets of images. For instance, one can divide a logical grid into arbitrary subgrids. Teams are of a special importance for PGAS models because this concept enables the partial allocation of memory in a subset of images and not along all the images. In this paper, we describe our design and implementation of coarray allocations, RMA, barriers, atomics and collective operations for image teams within the OpenUH compiler. We believe our platform represents the first compiler that supports the core features of teams expected in Fortran 2015.

To effectively make use of teams, images are logically grouped to work, in parallel, on the subproblems of an application. It is expected that these subproblems may be executed independently or close to independently; i.e., there should be minimal communication required between images executing in different teams. The language does not expose any other memory hierarchy information to the programmer, such as which images may be executing on the same compute node, with a more tightly coupled physical memory, versus a different node with loosely-coupled memories and higher communication costs. Therefore, among images executing as part of the same team, the amount of explicit control the programmer has in reducing communication cost is limited to reducing accesses of data belonging to a different image or reducing the occurrence of operations with implicit collective communication.

The second decomposition dimension we introduce in this paper relies on a specific hardware-aware dedicated runtime that can take advantage of the actual memory hierarchy in order to optimize communications among teams by distinguishing between intra-node and inter-node memory accesses. Here, our approach consists on identifying the images that run on the same node, within a team, assigning a leader for them and handling them using an intra-node strategy. Afterwards, the leaders, which are by definition on different nodes, are handled differently. The challenge here also is to define, for each collective operation, which algorithm is suitable for intra-node local memory accesses and which is the best for inter-node remote memory accesses.

We applied this methodology to three common operations which may be executed collectively by a team of images in CAF: barrier, all-to-all reduction and broadcast. The classical algorithms for all three of these operations entail a fixed communication pattern among the images, and consequentially their total communication cost is sensitive to the placement of the images in the parallel system. Since a significant part of the execution time of an application is consumed while waiting on completion of these operations, ensuring their implementations are efficient irrespective of image placement is paramount. In this paper, we exploit the knowledge of the architecture that a run-time system can have, and describe how we applied inside every team a new, two-level algorithm, called *Team Dissemination Linear Barrier* (TDLB), for the barrier collective operation. In this algorithm, synchronization among leaders of a node is performed with the dissemination algorithm, while synchronization within a node uses a linear barrier. Experimental evaluations indicate that the application of our proposed runtime awareness approach of the memory hierarchy to the implementation of barriers via TDLB yields an up to 26-time execution time improvement over the basic dissemination algorithm. We also applied this methodology to all-to-all reduction and one-to-all broadcast operations, but, for lack of space, we do not provide a detailed description of our two-level reduction and broadcast algorithms in this paper.

The contributions of this paper are thus:

- the support of teams in Coarray Fortran within the OpenUH compiler, enabling the creation of logical image subsets that work on loosely-coupled subproblems within an application. Specifically, we describe our original memory management strategy for allocation and deallocation within teams;
- the design of a two-step methodology for achieving better performance of collective operations, using runtime awareness of the memory hierarchy;
- the application of this methodology to the implementation of barriers, via the new Team Dissemination Linear Barrier (TDLB) algorithm, reductions and broadcasts: the novelty in these algorithms is to adapt existing techniques such as the dissemination algorithm to the PGAS memory model using one-sided communications;
- a comprehensive evaluation of this methodology on two benchmarks: (1) our newly developed Coarray Fortran, CAF 2.0 and MPI (communicator concept) Teams Microbenchmark suite [3], which contains a set of kernels for testing reductions, broadcasts and barriers within teams [4], and (2) our porting to Coarray Fortran of the High Performance Linpack (HPL) benchmark [4], which uses teams.

The paper is organized as follows. We describe the OpenUH compiler and Coarray Fortran in Section II. The design and implementation of teams of images within

[3]As of this writing, these features are described in N2040.pdf at http://www.nag.com/sc22wg5/.

[4]Since teams are a relatively new concept for Coarray Fortran, there is no reference test suite for them; Teams Microbenchmarks has been made publicly available for other implementers to get a baseline to compare themselves to.

the OpenUH compiler are introduced in Section III. In Section IV, we define our memory hierarchy awareness methodology and apply it to barrier, reduction and broadcast operations. Experimental results using our microbenchmarks and High Performance Linpack (HPL) are discussed in Section V. We survey other approaches to the implementation of teams in Section VI. We discuss future work and conclude in Section VII.

II. Background

We provide here the necessary background material for our work: the OpenUH compiler and Coarray Fortran.

A. The OpenUH Compiler

OpenUH [5] is a branch of the open-source Open64 compiler suite that has been developed at the University of Houston and is used to support a wide range of research activities in the area of programming model research. OpenUH provides a solid base infrastructure for exploring implementation strategies for Coarray Fortran. The Fortran 95 front-end, originating from Cray, was already capable of recognizing coarrays and parsing the cosubscript syntactic extension. OpenUH also includes its own Fortran runtime libraries, providing optimized support for the intrinsic routines defined in Fortran, memory allocation, I/O, and termination.

B. Coarray Fortran

Coarray Fortran is an explicitly-parallel extension of the Fortran 2008 standard that adheres to the Partitioned Global Address Space (PGAS) programming model. Coarray Fortran programs follow an SPMD execution model, where all execution units called images are launched at the beginning of the program; each image executes the same code and the number of images remains unchanged during execution.

1) Coarrays

Coarrays are shared data entities that are declared with the codimension attribute specifier and allocated collectively across all images. Coarrays are replicated a fixed number of times. Subscripts of coarrays are specified with square brackets and provide a clear and straightforward representation of access to data on other images using 1-sided communication semantics. One can specify 1-sided communication using coarrays. For example, the statement `A(:)[k] = B(:)` writes the elements of Coarray `A` on Image `k`.

2) Team-Based Clustering of Images

Teams, added to Coarray Fortran, induce a hierarchical SPMD model. The initial team contains all the images. Subsets of images in a team may collectively form a new team, which are referenced using a handle of type `team_type`, using the `form team` statement. Every team has a unique identifier and a unique parent. An executing image can change to a subteam of the current team using the `change team` construct. Statements `change team` and `end team` delimit a structured block while setting the team parameter within its scope. Teams can be used to partition an application into different tasks executed by subteams. For instance, one can divide a logical grid into arbitrary subgrids. This could be used to group subsets of images performing computations on dense matrices into row- and/or column-oriented teams.

Regarding performance, via teams many collective operations can be overlapped; these collectives will work on just a subset of images and an image need not be communicating or synchronizing with images belonging to other teams. This removes the need for global synchronizations among all the images. Regarding memory, using Coarray Fortran teams one can declare and allocate coarrays within a `change team` block. This allows a coarray to be allocated only in the images operating on it, thus utilizing more efficiently the available memory on each image.

III. Team Support in OpenUH

OpenUH is able to parse the `form team`, `change team`, `end team` and `sync team` constructs. We added the new type `team_type` to the type system of OpenUH and support for `get_team` and `team_id` intrinsics. We also extended the usual CAF intrinsics `this_image`, `num_images` and `image_index` for teams. We depict the OpenUH Coarray Fortran implementation in Figure 1.

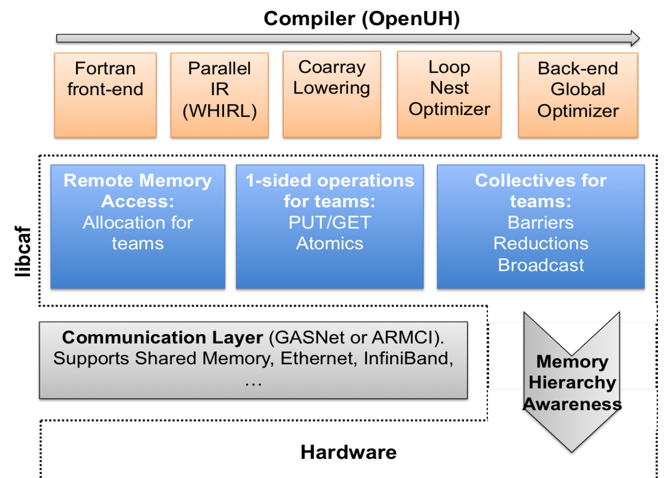


Figure 1: OpenUH Coarray Fortran team implementation

During the back-end compilation process in OpenUH, team-related constructs are lowered to subroutine calls which constitute the UHCAF runtime library interface. In the runtime, we add a `team_type` data structure for storing image-specific identification information, such as the mapping from a new index to the process identifier in the lower communication layer. Also we provide support for team-related intrinsics, for example `get_team` and `team_id`. We evaluate our implementation of teams and compare it to CAF 2.0 and MPI in Section V-B.

Before team support was added into our implementation, coarrays allocation was globally symmetric across all images, with each coarray allocated at the same offset within a managed symmetric heap. With teams, however, this global symmetry can be relaxed. According to the draft of the

technical specification, symmetric data objects have the following features, which simplify the memory management for teams. First of all, whenever two images are in the same team, they have the same memory layout. Second, an image can only change to the initial team or teams formed within the current team. Third, when exiting a given team, all coarrays allocated within this team should be deallocated automatically. And fourth, if an image wants to refer to a coarray of another image located in a sibling team, the coarray should be allocated in their common ancestor team.

A. *Memory Management*

As currently specified, coarrays must be symmetric across all images. Yet, a coarray may indirectly point to non-symmetric data, by declaring the coarray to be of a derived data type with a pointer or allocatable component, data for which may be allocated independently of other images. The allocated data may be remotely referenced using the coarray. In our implementation, these kinds of data objects are allocated in an *asymmetric* memory section.

We implemented a structure for managing allocations from a remotely accessible memory heap allocated during program initialization. We use this structure to manage both symmetric (i.e., coarrays) and non-symmetric (e.g., associated data of a coarray pointer component, or intermediate local communication buffers) allocations. The structure was implemented as a doubly-linked list of *slots*, each of which describes a range of addresses which may be either allocated or non-allocated. The slots list is ordered based on the address range referenced by each slot. The allocation strategy is to efficiently make use of the available space in the managed heap. This is achieved by allocating from the top of the heap for symmetric allocations and from the bottom of the heap for non-symmetric allocations. One node in the slots list, designated the *common slot*, serves as the last available slot for either symmetric or non-symmetric allocations. When an object is allocated from this heap, we search for an empty slot referencing an address range of sufficient size. We then either use this slot or split the slot in two if the address range exceeds the amount of memory being requested. When an object allocated from this heap is deallocated, the corresponding slot is marked as *empty* and merged with empty slots that may immediately precede or follow it in the slots list.

When a coarray is allocated while executing a `change team` block (see for instance Figure 2), corresponding allocations should occur on all other images in the *current* team, rather than for all images. Upon exiting the `change team` block, any allocations that had occurred within it are implicitly freed if they were not already freed by a `deallocate` statement. An image may only *change* to a team with the `change team` construct if it was formed by its current team with a `form team` statement or if it is the initial team. The latter scenario requires that the state of symmetric allocations belonging to the initial team should not be affected by allocations (not yet freed) belonging to a non-initial team. To support this, we reserve a fixed section of memory from the top of our managed heap for symmetric allocations by a non-initial team.

We divide the list structure for memory allocations into two lists: one is for symmetric allocations by any non-initial team, and the other is for symmetric allocations by the initial team and all non-symmetric allocations. Whenever the image changes to a team, the old heap address within the segment for that team is saved in the team structure as `symmetric_slot`; when the team ends, this data is easily freed. Section IV-B presents the team data structure.

```
change team(cteam)
  ...
  allocate(w(nn+1, BLKSIZE)[0:*], wptr(1)[0:*])
  ...
  mykey = 1
  ...
  form team(mykey, subcteam)
  change team(subcteam)
    ...
    call co_broadcast(u, 1)
    ...
  end team
  ...
  deallocate(wptr, w)
  ...
end team
```

Figure 2: HPL code snippet from our Coarray Fortran implementation using teams

B. *Remote Memory Accesses*

To refer to an object in another image's memory, we use a "`base_address` plus offset" technique to compute the remote data address. The base address is obtained during initialization. From the discussion above, the memory layout symmetry between images in the same team is preserved. So we still can compute the remote address of a coarray in this simple manner.

In CAF, a coarray may have the `save` attribute, or is allocated by images executing as part of some team (the initial team, or a formed team during execution of the `change team` construct). Our memory allocation scheme guarantees that a coarray will reside at the same offset within the managed symmetric heap as the corresponding coarray of any other image in which it was allocated, with the base address of the symmetric heap being set during program initialization. Hence, accessing a team-allocated coarray on another image entails no additional overhead compared to accessing a save coarray or a coarray allocated by images in the initial team.

C. *Collective and Atomic Operations*

We adapt atomic operations (`atomic_add`, `atomic_and`, etc.), synchronization operations (`sync images` and `sync all`), broadcast (`co_broadcast`)

and reduction operations (`co_sum, co_max, co_min`) to work when executed by non-initial teams. Each subroutine works using the global pointer to the current team, quickly obtaining the mapping of the image ids to the process ids in the `team_type` structure's image index mapping array.

IV. Memory Hierarchy and Teams

In order to make applications more scalable when running on nodes with many cores, the runtime should have some knowledge about the mapping of images on nodes and/or cores. If teams create subsets of images, there is no simple relationship between the image structure and the actual underlying physical structure of the parallel system. Therefore, as a research methodology towards an efficient implementation of teams, we propose to introduce a memory hierarchy-aware runtime for PGAS, in order to optimize communications within teams via the distinction between local and remote memory accesses. In this section, we present our methodology and its application to the barrier operation through the use of a two-level barrier algorithm. As already mentioned, we also applied this methodology to reduction and broadcast operations but left the description of these other use cases out this paper, for lack of space.

A. Methodology

A major motivation for applying our methodology to barriers is that the classic dissemination barrier algorithm is well-suited for distributed memory systems but not as efficient for the shared memory case. In the dissemination algorithm, for n images, there are $n \log n$ synchronization notifications. On a shared memory system, where the n processors share the same physical memory, in the worst case all those notifications would have to be serialized. However, contrast this with a centralized linear algorithm. For n processors, there are $2(n-1)$ notifications because there are two steps: first, notifications are sent from $n-1$ non-leader images to the leader image; then, in the second round, notifications proceed from the leader image to the $n-1$ slave images. Even if all those notifications are serialized, it is not as expensive as for the dissemination algorithm. If we consider instead a distributed system, where each of the n images is on its own node, then dissemination becomes faster. There are $n \log n$ total notifications, with n notifications performed in parallel in $\log n$ steps. For a centralized linear algorithm, everything would have to be serialized through a single node, so yielding $2(n-1)$ steps. These results are confirmed by Mellor-Crummey *et al* [6].

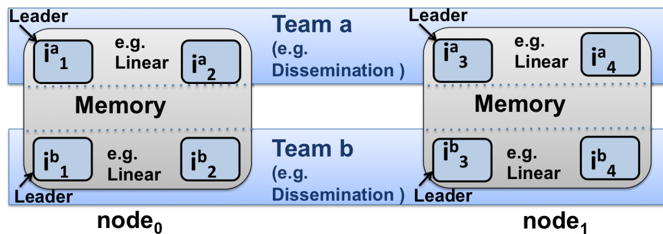


Figure 3: Memory hierarchy awareness methodology for two teams; Image i^t_x denotes the xth image of the tth team

Our methodology relies on detecting the images within a team that run locally on the same node, assigning a leader for them and handling them with an intra-node strategy. After that, the leaders, which are on different nodes, are handled in a remote manner. This methodology is presented in Figure 3.

B. Team Data Structure

In our approach, we combine the grouping of images into teams with a memory hierarchy-aware runtime. Thus, in our design of teams, we use a team data structure that includes data objects related to teams and to memory hierarchy. Also, in order to make our implementation the fastest possible, we compute the information related to a team once each time we form a new team of images in order to avoid its calculation at every collective operation on a team.

Table I: Team data structure

Feature	Meaning
parent	Parent team in the tree of created teams
num images	Number of images in this team
depth	Level of nesting of this team in the team tree
image index mapping	Correspondence between the logical image id in a team and the hardware entity process id
intranode set	Set of images on the same node within the same team
leaders set	Set of leaders among nodes within the same team
sync flags	Array of synchronization flags needed for implementing dissemination barrier
sync parity	Parity variable controls the use of the sync flags in successive barriers
sync sense	Phase of the synchronization
cocounter	Counter variable when applying a counter-based algorithm for barrier
symmetric slot	Heap address for the symmetric memory slot reserved for this team

The information we need to store includes (1) the list of images that are on the same node, (2) the number of images within the same node, and (3) data structures used to facilitate the collective operations. This information is used many times in the runtime by different algorithms to perform collectives (see Section IV-C). Table I summarizes the features required for each team and their meaning.

C. The TDLB Synchronization Algorithm

In order to implement a memory hierarchy-aware barrier, we developed a two-level algorithm that proceeds in three steps: (1) a designated leader on each node waits for the remaining images on the same node to arrive at the barrier; (2) all leader images, one from each node with at least one image in the team, synchronize using a dissemination barrier algorithm; and (3) each node leader notifies the remaining images on the same node that they may leave the barrier.

As we said, the dissemination algorithm is suitable for message passing, but tends to perform poorly in the shared

ALGORITHM 1: Team Dissemination Linear Barrier Algorithm, to be run by each image in Team `team`

```
procedure TDLB(team)
 me = this_image(team)
 cocounter = team.cocounter
 ι_leader = get_leader(team, me)
 //step 1: slaves synchronize with the leader
 linear_counter_step1(team,me,ι_leader,cocounter);
 if (ι_leader == me) then
  pgased_dissemination(team, ι_leader);
 //step 2: the leader notifies the intranode set
 linear_counter_step2(team,me,ι_leader,cocounter);
 end
```

memory case. A centralized linear algorithm, such as a counter-based algorithm, is the best way to program a shared memory barrier for a limited number of images. Therefore, in our implementation, synchronization among nodes' leaders is performed using the dissemination algorithm, while synchronization within a node uses a linear barrier. This algorithm is called Team Dissemination Linear Barrier Algorithm (TDLB), and is specified in Algorithm 1. In this algorithm, there is an interaction between two barrier algorithms. Note that, in the subsequent algorithms, we use the same syntax as Coarray Fortran to access an element of an array, using parentheses as array subscripts and brackets for image-selecting cosubscripts.

D. Cost Model

We present a simple cost model to estimate the total time taken by TDLB based on two parameters: latency (internode α_n, intranode α_c) and bandwidth (internode β_n, intranode β_c). Let $L(t)$ be the set of nodes containing at least one image that belongs to a team t. Within t, we select one image leader $l(t, n)$ for each node $n \in L(t)$. $|L(t)|$ is thus the number of leader images within Team t. We note $I(t, l)$ is the set of images under one leader l within t. This cost model assumes that all images can *put* and *get* one message at the same time and all local nodes are strictly flat.

1) Dissemination for Remote

Algorithm 2 describes an adaptation of the dissemination algorithm described in [6] to the use of teams in a PGAS environment. There, the term *sense* refers to a phase of the synchronization process. The sense variable `sync_sense` ensures that the participating images are in the same phase of the various synchronization steps in the whole program. It is a boolean variable that alternates its value from one barrier to the next. Each image maintains a sense variable for the other images (`target`). In [6], the partner flags are gathered in a separate array in a shared memory algorithm; in our PGAS case, the concept of 1-sided communication is used to refer to the sync flags of the target at each step via `sync_flags(...)[target]`. `sync_flags` is a two-dimensional coarray of (2,$\log_2(|L(t)|)$) elements used to synchronize the $|L(t)|$ images along all the $\log_2(|L(t)|)$

ALGORITHM 2: PGASed Dissemination Barrier Algorithm for Team `team`, run on the ι_{leader} image of a node

```
procedure pgased_dissemination(team, ι_leader)
 leaders_set = team.leaders_set;
 sync_flags = team.sync_flags
 sync_sense = team.sync_sense
 sync_parity = team.sync_parity
 rank = get_leader_rank(leaders_set, ι_leader);
 nbsteps = log2(|leaders_set|);
 sense  = 1 - sync_sense; parity = sync_parity;
 //step 1: sync with the other leaders
 for(step = 0, nbsteps -1)
  power = 2^step;
  target = image_id((rank + power)%count,team);
  source = image_id((rank - power + count)%count,
                                             team)
  sync_flags(parity, step)[target] = sense;
  block_until(sync_flags(parity,step) == sense);
 //step 2: toggle the sense
 sync_parity = 1 - parity;
 if (sync_parity == 1) then
  sync_sense = sense;
end
```

steps of the dissemination algorithm. The first dimension represents the parity variable that controls the use of alternating sets of flags in successive barriers. *false* is the initial value of the flag, and it means the image has not arrived yet to the barrier, and *true* means that the image has reached the barrier. `target` is the image that an image ι_{leader} needs to synchronize with during a given step. `source` is the image ι_{leader} is waiting for. Function `get_leader_rank` returns the position of ι_{leader} in `leaders_set`; this position needs to be 0-adjusted since in Fortran indices typically start from 1. Also, `image_id` returns an image index with respect to the initial team, based on a 0-adjusted index with respect to another team. Function `block_until(cond)` is used to block until the value of `cond` is true. At every step, each image sends a notification to the target image, and thus $|L(t)|$ notifications occur at every step. The time taken by this algorithm for Team t is:

$$T_{dissemination}(t) \quad = \quad \log(|L(t)|)\alpha_n + |L(t)| \log(|L(t)|)\beta_n$$

2) Parallel Linear for Local

We deal with images on the same node via a linear algorithm, a counter-based one. In the counter-based algorithm presented in Algorithm 3, we use a coarray `cocounter` for each team to synchronize the non-leader images ι and the leader ι_{leader}. In the first step, each non-leader image ι remotely writes 1 to the `counter` of the leader image ι_{leader}, at the position ι, and then waits. In the second step, as shown in Algorithm 1, once ι_{leader} has synchronized with the other leaders, it remotely writes 1 back to the `cocounter` of the slave images in order to release them. Function `changed_p` is used to return *true* if the value of `cocounter` for the corresponding image has changed.

ALGORITHM 3: Steps 1 and 2 of Parallel Counter-Based Barrier Algorithm, run by each image ι in Team `team` with Counter `cocounter`; each image has a leader, ι_{leader}, on the same node

```
procedure linear_counter_step1(team, ι, ι_leader,
                                              cocounter)
 if (ι_leader == ι) then
  foreach ι_i ∈ intranode_set(team, ι_leader)
   block_until(changed_p(cocounter(ι_i)));
   cocounter(ι_i) = 0;
 else
  cocounter(ι)[ι_leader] = 1;
end
procedure linear_counter_step2(team, ι, ι_leader,
                                              cocounter)
 if (ι_leader == ι) then
  foreach ι_i ∈ intranode_set(team, ι_leader)
   cocounter(ι_leader)[ι_i] = 1;
 else
  block_until(changed_p(cocounter(ι_leader)));
  cocounter(ι_leader) = 0;
end
```

This linear algorithm is an optimized, counter-based algorithm because the counter `cocounter` here is implemented with an array rather than a scalar, unlike what is usually used in linear algorithms, in order to make it possible for notifications from the participating images to be received in parallel. If we had used a scalar, the updates would have been serialized via an atomic increment operation. The number of notifications in the optimized version is thus $|I(t,l)|$ instead of $2(|I(t,l)|-1)$. This approach improves performance at the cost of an extra byte for each additional non-leader node on the same image.

The linear algorithm is structured in two steps. First, all images send in parallel $|I(t,l)|-1$ notifications to the leader coarray, which has cost 1, since this happens in parallel. In the second step, the leader sends $|I(t,l)|-1$ notifications to the images. So the time taken by these two steps is:

$$T_{linear}(t,l) \quad = \quad 2\alpha_c + |I(t,l)|\beta_c$$

3) Total Cost

Recall that $L(t)$ is the set of leader images within a team t and $I(t,l)$ the set of images under one leader l within a team t. The time taken by the Team Dissemination Linear Barrier Algorithm operating on a team t is thus:

$$\begin{aligned} T_{TDLB}(t) \quad &= \log(|L(t)|)\alpha_n + |L(t)|\log(|L(t)|)\beta_n + \\ &\sum_{n\in L(t)}(2\alpha_c + |I(t,l(t,n))|\beta_c) \end{aligned}$$

V. Experimental Results

This section discusses experimental results for our approach on two benchmarks: (1) the Teams Microbenchmark suite [3], which contains code designed to test the performance and correctness for collective operations (reduction, broadcast and barrier) and team operations (team formation and team switching) written in CAF, CAF2.0 and MPI, and (2) a porting of High Performance Linpack (HPL) to use Fortran 2008 coarrays and anticipated future constructs (teams and collective subroutines).

A. Experimental Setup

We ran our experiments on a cluster of 44 nodes connected via a 4xDDR InfiniBand (IB) switch, with dual quad-core AMD Opteron processors running at 2.2GHz on 16GB of main memory per node. We compared the implementation of these benchmarks in OpenUH 3.0.40 with two other implementations we used for this evaluation: (1) a CAF 2.0 version, for the source-to-source Rice CAF 2.0 compiler version 1.14.0, which uses ROSE [7] and GFortran 4.4.7 as backends, and (2) an MPI version, which we ran using both Open MPI 1.8.3 and MVAPICH 2.0beta. Both OpenUH and Rice CAF 2.0 implementations rely on GASNet's Infiniband Verbs runtime implementation. We used GASNet 1.22.2.

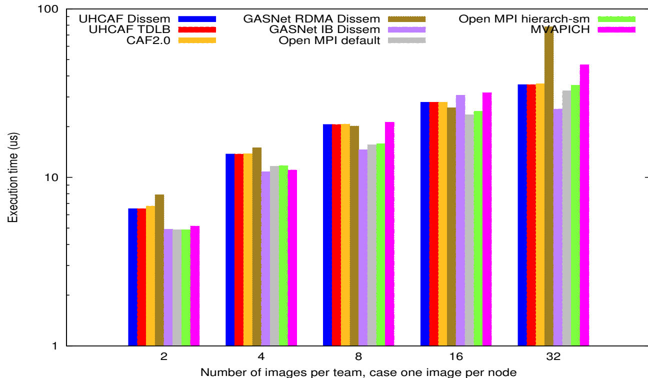
B. Teams Microbenchmarks

In this section, we compare the performance of our implementation for teams with the Rice CAF 2.0, Open MPI and MVAPICH implementations. We applied the methodology introduced in this paper to barrier, all-to-all reduction and one-to-all broadcast operations. The performance of these collectives are assessed via our microbenchmarks.

Barrier Contrarily to Algorithm 9 in [6], which relies on two synchronization arrays for its implementation of a barrier operation, and the one described in [1], which is using two waits, our dissemination algorithm is based on Coarray `sync_flags`, thus taking advantage of the features of a PGAS model with only one wait. We compare our TDLB implementation of barriers with the one provided on GASNet, which implements very low-level algorithms depending on the conduit used. TDLB is portable and can be used with any communication layer and any conduit. Our methodology tends to be independent from the low-level communication layer used to implement PGAS languages/libraries.

In the two charts in Figure 4, the RDMA Dissemination uses Put operations to implement the dissemination barrier described in [6]. The IB Dissemination directly uses Infiniband verbs for communication to implement the same algorithm. CAF 2.0 uses also the dissemination barrier described in [6]. As for MPI, we use `MPI_Barrier` of MVAPICH, default Open MPI and Open MPI with the hierarchy-awareness options (`hierarch` and `sm` modules). In UHCAF, we compare TDLB with the pure dissemination algorithm described in [6]. We use two configurations: (1) one image per node, to verify that TDLB performs as well as a pure dissemination algorithm in the case of a flat hierarchy, and (2) 8 images per node, to show that TDLB is well optimized to handle the memory hierarchy and is only marginally more expensive than the low-level dissemination algorithm implemented directly over the IB verbs that GASNet provides.

All-to-All Reduction In the two charts in Figure 5, we compare the application of our methodology on the reduction operation to the original implementation, which



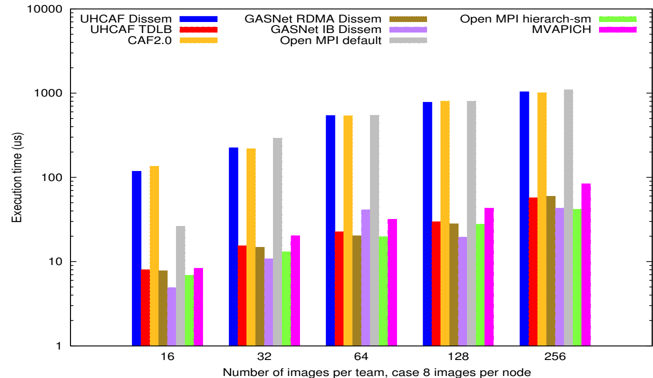


Figure 4: Performance evaluations for TDLB barrier algorithm using the Teams Microbenchmark suite

uses the recursive doubling algorithm [8]. The two-level implementation uses binomial tree reduction from non-leaders to their node leader; then, a recursive-doubling all-reduce is performed between the leaders; and finally the non-leaders perform parallel local gets from their leader. We also compare it to CAF 2.0, Open MPI and MVAPICH. As expected, the memory hierarchy awareness in our two-level algorithm gives very good results (note the case where we have 8 images per node).

In the case of one image per node, not only is there no additional overhead compared to the original implementation, but we were able to improve the performance by applying a further optimization. Using the 2-level approach advocated in our paper, we can distinguish remote memory operations that access out-of-node memory via the interconnect's RDMA from memory accesses within the node. In the former case, we can employ the canary protocol [9], which entails the target polling on the last byte (or some bytes) as a canary value to check for communication completion (a valid approach because an RDMA write over Infiniband can be assumed to complete in byte order). By using this protocol, which effectively bundles a notification of completion with the data to be sent, we can eliminate sending an additional notification per write in our algorithm.

One-to-All Broadcast We also applied our 2-level methodology to broadcast operations. The two charts in Figure 6 show results comparing our implementation of the two-level broadcast in UHCAF, with Open MPI, MVAPICH and Rice CAF 2.0 versions. Our algorithm proceeds in three steps: (1) we first write to the leader image on the first node; (2) we use a binomial tree broadcast among the leader images at each node with the source image being the leader at the first node; and (3) the non-leaders at each node wait for a notification from their leader and then read the result from the leader's buffer. We compare this broadcast with three MPI broadcast (`MPI_Bcast`) implementations: Open MPI's default broadcast, Open MPI broadcast with hierarchy-awareness (the MCA `hierarch` and `sm` modules) enabled, and MVAPICH's broadcast. For UHCAF, we compare the two-level implementation with a 1-level binomial tree broadcast. We use two configurations for our comparisons: one image per node and 8 images per node. Again, by breaking up into inter-node and intra-node broadcast, we can optimize inter-node communication with by using the canary protocol, referenced above. Our two-level algorithm is well optimized to handle the memory hierarchy, and is faster that CAF 2.0, Open MPI and MVAPICH.

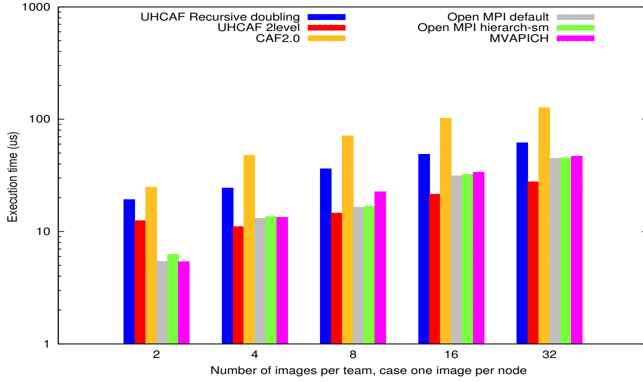
C. *HPL*

For the purpose of this work, we implemented a Coarray Fortran version of High Performance Linpack (HPL) [4], which is used to solve systems of linear equation, thus testing temporal and spatial run-time locality. We based our version of HPL on its CAF 2.0 port, described in [10]. HPL makes use of row team *rteam* and column team *cteam* for performing updates of the matrix data. HPL computes LU factorization with row partial pivoting. The recursive factorization of a 1-dimensional panel of columns of processes is performed on *cteam*. The associated swaps and broadcasts of the pivot row are performed on *rteam*. Figure 7 compares the performance results using the two-level approach in UHCAF, the one-level approach in UHCAF, CAF 2.0 using GFortran as backend compiler, CAF 2.0 using OpenUH as backend compiler and Open MPI using GCC compiler. The `-O3` option is passed to the compilers in Figure 7. These preliminary results show that using the two-level approach in UHCAF provides up to 32% improvement over a typical one-level approach. Overall, we obtained 95 GFLOPS/s on 256 cores, as compared to 29.48 GFLOPS/s obtained with the CAF 2.0 implementation with the GFortran backend and 80 GFLOPS/s with the OpenUH backend. Note that we used the classic recursion parameters of the MPI version of HPL without tuning; we got results comparable to UHCAF's.

VI. Related Work

In this section, we survey different existing implementations of teams and compare them with our approach.

CAF 2.0 [11], a alternative Fortran extension for coarrays proposed by Rice University, included teams as first-class objects since its inception. It provides similar operations



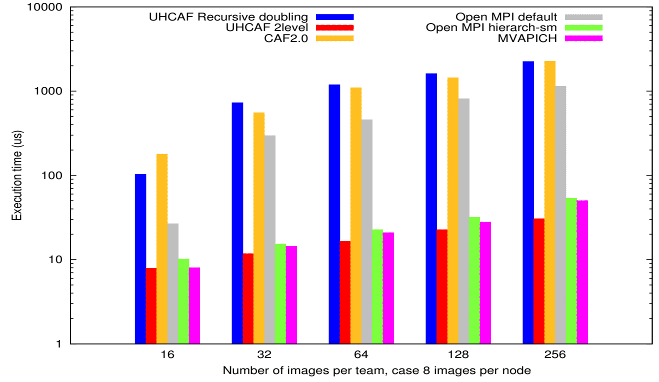
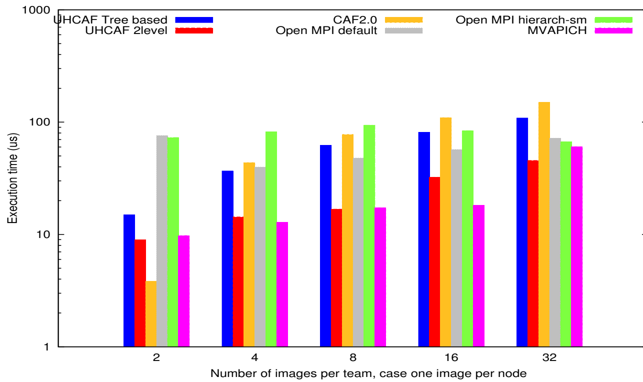


Figure 5: Performance evaluations for the 2-level reduction algorithm using the Teams Microbenchmark suite



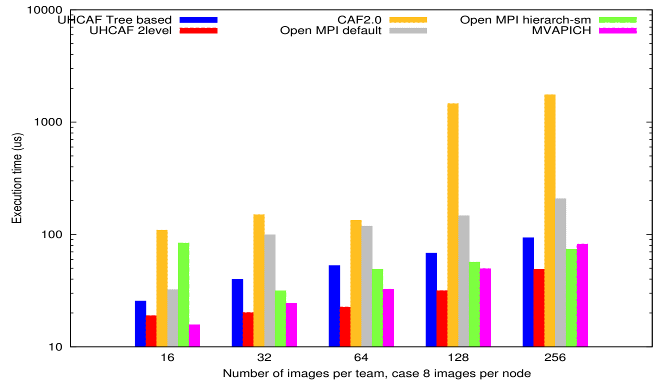


Figure 6: Performance evaluations for the 2-level broadcast algorithm using the Teams Microbenchmark suite

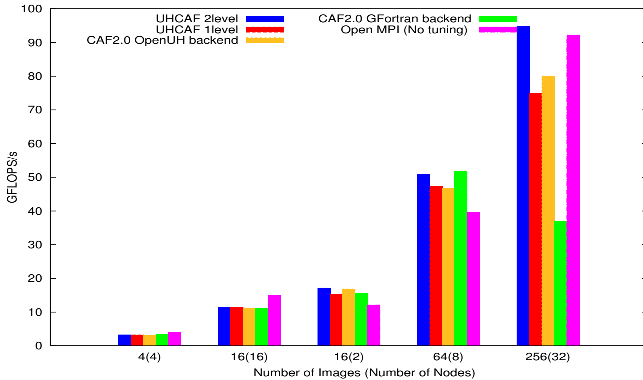


Figure 7: Performance results for HPL

as what is being proposed for Fortran 2015. CAF 2.0 also permits a more unstructured programming style for team-based collectives, allowing an image to execute a collective operation (e.g. reduction) as part of a specified team without changing the current team. This added flexibility in the language, however, can introduce difficult-to-detect synchronization bugs into the code, and can impose additional synchronization costs for completing collective operations.

OpenSHMEM [12] proposes the concepts of teams and spaces in order to allow allocation of memory only across subteams. The referenced paper explores the allocation, placement and access to data by subgroups of processes and studies the positive impact of teams on the symmetric memory requirements for OpenSHMEM codes.

These two works, which extend Fortran and OpenSHMEM implementations with team support, did not provide any memory hierarchy information for the teams. In our implementation, we combine both team-based grouping of images with an awareness of the underlying memory hierarchy, lacking in the referenced implementations of OpenSHMEM and CAF 2.0.

MVAPICH2-X [13] provides a unified runtime that supports both MPI and PGAS programming models, namely OpenSHMEM and UPC on IB clusters. The referenced paper proposed a team-based memory allocation strategy and introduces `shmem_team_create` and `shmem_team_split`, which are MPI-like routines to support SHMEM extensions for creating and splitting teams in order to define symmetry domains for symmetric allocation. UHCAF, in addition, provides an extension for both memory and collective operations to work in the context of teams.

The notion of grouping and hierarchy is present in other programming paradigms. For instance, the Message Passing Interface-2 (MPI-2) specification allows for remote memory access (RMA) within a group of MPI processes represented by a communicator through the mechanism of window

creation [14]. The routine `MPI_Comm_split` partitions the group associated with an initial communicator into disjoint subgroups, one for each value of color. Techniques to support the scalability of communicators and groups in MPI are presented in [15]. Our work parallels this approach, within the PGAS framework. We believe that the team-based, one-sided communication model of Coarray Fortran is easier to handle by programmers than the grouping semantics of MPI (and its need for windows); our work shows that this better programmability does not come at a cost, since our implementation of Coarray Fortran is competitive with MPI.

Finally, note that Open MPI provides two options – hierarch and sm – which we enabled for our comparisons to make Open MPI collectives work in a multi-level fashion (see Section V). Our understanding is that Open MPI makes use of inter-node and intra-node communicators, applied respectively in two separate steps. In our approach these two steps are more integrated and may be partially overlapped due to the use of 1-sided communication (e.g., while communication is occurring among leaders, the non-leaders may proceed with the next collective call).

VII. Conclusions and Future Work

In this paper, we proposed a PGAS-based design methodology for supporting efficient communication between teams for Coarray Fortran that takes into account the memory hierarchy of clusters to efficiently implement collective algorithms. We showed how the memory hierarchy can be exploited to optimize team implementations via the distinction between local and remote memory accesses. Specifically, we focused on reducing the cost associated with moving data within a node and between nodes. This hardware-aware approach used within the runtime system leads to a significant optimization of the communications within the virtual topologies and between the clusters that are binding different teams together.

To evaluate our proposal, we extended the implementation of Coarray Fortran within the OpenUH compiler with teams and applied our methodology to three important collective operations: barrier, reductions, and broadcast. The use of our memory hierarchy-awareness approach for these operations shows up to, respectively, 26-, 74- and 3-fold performance improvements over the default approach. We also evaluate our two-level methodology on HPL and get better performance results compared to the one-level approach and original CAF 2.0 version we based ours on.

Future work will look at how our methodology can support multi-level hierarchies to represent different network topologies or on-node locality domains such as NUMA memory nodes, shared caches, processor sockets and cores.

References

[1] D. Hensgen, R. Finkel, and U. Manber, "Two Algorithms for Barrier Synchronization," *Int. J. Parallel Program.*, Feb. 1988.

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

[3] "HPCTools Teams Microbenchmarks," https://github.com/dkhaldi/teams_microbenchmarks.

[4] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/.

[5] B. Chapman, D. Eachempati, and O. Hernandez, "Experiences Developing the OpenUH Compiler and Runtime Infrastructure," *Int. J. Parallel Program.*, vol. 41, no. 6, pp. 825–854, Dec. 2013.

[6] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-memory Multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, Feb. 1991.

[7] D. Quinlan, "ROSE: Compiler Support For Object-Oriented Frameworks," *Parallel Processing Letters*, 2000.

[8] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[9] T. Hoefler and T. Schneider, "Optimization Principles for Collective Neighborhood Communications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012.

[10] J. Guohua, J. Mellor-Crummey, L. Adhianto, W. Scherer, and C. Yang, "Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 1089–1100.

[11] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, and G. Jin, "A New Vision for Coarray Fortran," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, ser. PGAS '09. New York, NY, USA: ACM, 2009, pp. 5:1–5:9.

[12] A. Welch, S. Pophale, P. Shamis, O. Hernandez, S. Poole, and B. Chapman, "Extending the OpenSHMEM Memory Model to Support User-Defined Spaces," *PGAS 2014*, oct 2014.

[13] J. Jose, K. Hamidouche, X. Lu, S. Potluri, J. Zhang, K. Tomko, and D. K. Panda, "High Performance OpenSHMEM for MIC Clusters: Extensions, Runtime Designs and Application Co-design," *IEEE Cluster 2014*, sep 2014.

[14] A. Moody, D. Ahn, and B. Supinski, "Exascale Algorithms for Generalized MPI_Comm_split," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2011.

[15] H. Kamal, S. M. Mirtaheri, and A. Wagner, "Scalability of Communicators and Groups in MPI," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 264–275.