Evaluating Coarray Fortran with the CGPOP Miniapp

Andrew I. Stone
Colorado State University
stonea@cs.colostate.edu

John M. Dennis
National Center for Atmospheric Research
dennis@ucar.edu

Michelle Mills Strout
Colorado State University
mstrout@cs.colostate.edu

Abstract
The Parallel Ocean Program (POP) is a 71,000 line-of-code program written in Fortran and MPI. POP is a component of the Community Earth System Model (CESM), which is a heavily used global climate model. Now that Coarrays are part of the Fortran standard one question raised by POP’s developers is whether Coarrays could be used to improve POP’s performance or reduce its code volume. Although Coarray Fortran (CAF) has been evaluated with smaller benchmarks and with an older version of POP, it has not been evaluated with newer versions of POP or on modern platforms. In this paper, we examine what impacts using CAF has on a large climate simulation application by comparing and evaluating variants of the CGPOP miniapp, which serves as a performance proxy of POP.

1. Introduction
Large scientific simulation applications commonly use MPI to introduce parallelism and conduct communication. Although MPI is a mature and popular interface, developers often find it difficult to use. MPI requires programmers to handle a large number of implementation details such as the explicit marshalling and unmarshalling of data into messages and the explicit specification of communication schedules. MPI is commonly criticised as being low-level and is sometimes referred to as the “assembly language of parallel programming” [13].

Parallel programming models such as the PGAS [12, 26, 32] and DARPA HPCS [21] languages avoid explicit message passing and have been developed to make programming parallel applications easier. These languages have been shown to perform well and improve programmer productivity within the context of benchmarks and applications written from scratch [4, 19, 31]. However, the productivity of PGAS languages within large, existing, simulation codes has not been extensively researched.

The Parallel Ocean Program (POP) [20], developed at Los Alamos National Laboratory, is a large simulation code that runs on machines with thousands of cores. Much of POP’s complexity lies in handling parallelization and communication details. As such, it is worth considering whether a PGAS language could improve POP’s performance or reduce its code volume (total lines of code).

Given that POP is written in Fortran, using Fortran’s Coarray extensions is a logical step towards introducing a PGAS model. However, due to POP’s size and complexity it is desirable to avoid integrating Coarrays into the entire application until their benefit has been shown in a smaller prototype application. In this paper we use the CGPOP miniapp [28] as such a prototype. CGPOP models POP’s Conjugate Gradient routine and contains about 3000 source lines of code (SLOC) versus the 71,000 lines of POP.

During this investigation we developed several different variants of the CGPOP miniapp with the following questions in mind:

- How does the performance of the CAF variant of CGPOP compare with the original MPI variant extracted from POP?
- How will using an interconnect with direct PGAS support impact performance?
- Does transferring data in CAF by pulling (via get operations) differ in performance from pushing data (via put operations)?
- How easy is it to introduce a communication/computation overlap with the CAF version of CGPOP?
- What features are missing in the CAF standard and/or current implementation that are necessary to implement an efficient CAF version of POP or would otherwise be useful?

To answer these questions, we describe the CGPOP miniapp in Section 2. We show that CGPOP accurately models the performance of POP on two different Cray XT5 systems, a Cray XE6, and a BlueGene/L system, and describe several variants of the miniapp developed to compare CAF and MPI. In Section 3 we present how these different variants compare in terms of performance and code volume. In Section 4 we discuss our experience using CAF and document issues we encountered while programming with it. In Section 5 we discuss other work that compares MPI to PGAS languages, and in Section 6 we conclude this paper.

2. A Performance Proxy for POP
The Parallel Ocean Program (POP) has been actively developed for over 18 years [20]. Due to its continued use, maintenance of the application in terms of porting it to new architectures is an ongoing issue. Further, POP can be resource intensive to execute. While versions of POP at low-resolution exist, they do not exhibit the same sensitivity to communication performance as the versions at a 0.1° resolution. On lower memory systems like BlueGene the 0.1° POP execution requires a minimum of approximately 800 cores to execute. Even on systems with larger amount of memory per core a minimum of 96 cores is needed for as long as 40 minutes just to run a single simulated day. POP applications also includes a complicated build system that may have to be modified for a new compiler and support stack when moving to a new system. Clearly a much smaller, less resource intensive piece of code that serves as a proxy for the full application would enable a quicker turn around in the development cycle. Another advantage of a smaller proxy is...
that developers could prototype changes in it without changing the larger POP application.

We developed the CGPOP miniapp to serve as a proxy for POP. We started development of CGPOP in June 2010 and released version 1.0 of it in July 2011 [1, 28]. This section shows that CGPOP miniapp matches the performance profile of POP, defines what requirements variants of CGPOP miniapp fulfill, and describes the variants of CGPOP that we use to compare CAF and MPI.

2.1 CGPOP as a performance proxy

To be considered a performance proxy, a miniapp should accurately model the performance bottleneck of the full application at the range of cores that the full application targets. Since the POP application typically runs on thousands to tens of thousands of processors we compare the scalability of CGPOP and POP along this range.

Scalability can be affected by a number of factors including the machine and compiler used. To ensure that the performance behavior of the CGPOP miniapp matches that of POP we examine scalability across several different platforms: Hopper, a Cray XE6 located at the National Energy Research Supercomputing Center (NERSC); Frost, a BlueGene/L located at the National Center for Atmospheric Research (NCAR); Lynx [2], a Cray XT5 also located at NCAR; and Kraken, a Cray XT5 located at the National Institute for Computational Science (NICS). We list technical information about these compute platforms in Table 1. The compilers we used in our examination were PGI Fortran, Cray Fortran, and XL Fortran.

We present our results in Figure 2, and as can be seen by comparing the similarly colored lines for POP and CGPOP, the scalability behavior of the two are comparable when the same compiler and machine are used.

2.2 CGPOP miniapp specification

The CGPOP miniapp is defined in terms of its input/output behavior and the algorithm it conducts. We illustrate this behavior in Figure 1 and list pseudocode for the CG algorithm in Figure 6. As shown in Figure 1, the CGPOP miniapp executable is passed an intermediate state file, which is generated by the cginit domain decomposition generator. The cginit domain decomposition generator is passed an input file that contains stencil coefficients that are used with the discretization to construct the sparse matrix, a mask to indicate if a grid point is ocean or land, and the initial guess for vector x.

Table 1. Description of compute platforms used for this study.

<table>
<thead>
<tr>
<th>System</th>
<th>Name</th>
<th>Company</th>
<th>System Type</th>
<th># of cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Kraken</td>
<td>Cray</td>
<td>XT5</td>
<td>99,072</td>
</tr>
<tr>
<td></td>
<td>Hopper</td>
<td>Cray</td>
<td>XE6</td>
<td>153,408</td>
</tr>
<tr>
<td></td>
<td>Lynx</td>
<td>Cray</td>
<td>XT5</td>
<td>912</td>
</tr>
<tr>
<td></td>
<td>Frost</td>
<td>IBM</td>
<td>BG/L</td>
<td>8192</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>MHz</td>
</tr>
<tr>
<td>Peak Gflops/core</td>
</tr>
<tr>
<td>cores/node</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 data-cache</td>
</tr>
<tr>
<td>L2 cache</td>
</tr>
<tr>
<td>L3 cache</td>
</tr>
<tr>
<td>(shared)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Links/per node</td>
</tr>
<tr>
<td>Bandwidth/link</td>
</tr>
</tbody>
</table>

Figure 2. Execution time in seconds for 1 day of the barotropic component of the POP 0.1° benchmark and the 2-sided MPI version of the CGPOP miniapp on three different compute platforms.
and final solution for vectors $x$ and $b$ of Figure 6. The domain decomposition generator breaks an $3600 \times 2400$ array of ocean data into subdomain blocks that are distributed to each processor.

In addition to the data component of the intermediate state file, there is metadata that describe the relationship between blocks. There is a set of block information records, a graph of neighbors, and integer arrays that correspond to the global degree of freedom (GDOF) for every point in each block. GDOF values are identifiers that are unique for each grid point in the global domain. The block information records identify the location of each rectangular block within the global domain in terms of two-dimensional indices in the global domain.

After being generated, the intermediate state file can be passed to the CGPOP executable. After executing, CGPOP outputs correctness and timing data. Timing information is needed for evaluating performance. The correctness test, which is used for code verification, checks that the L2 norm for $x$ calculated by CGPOP matches that calculated by POP.

The algorithm CGPOP implements (see Figure 6) is a version of conjugate gradient that uses a single inner product [8], to iteratively solve for vector $x$ in the equation $Ax = b$. The matrix $A$, along with the initial guess vector $x_0$, right hand side vector $b$, and diagonal preconditioner vector $M^{-1}$ are read from an intermediate state file and passed as inputs into the function CGPOP-solver. The final surface pressure vector $x$ is the output of CGPOP-solver. The CGPOP-solver algorithm consists of a number of linear algebra computations interspersed with two communication steps. The GlobalSum function performs a 3-word vector reduction, while UpdateHalo function performs a boundary exchange between neighboring subdomains. The UpdateHalo function is passed an array that has been distributed across processes using the distribution described in the blocks data-structure of the intermediate state file. The asterisk * indicates a dot product between two vectors involving all entries in the local subdomain, and the GlobalSum results in the full dot product being completed.

2.3 Variants of CGPOP

We constructed several variants of the CGPOP miniapp to compare how Coarray Fortran and MPI can be used to implement communi-
Figure 6. CGPOP’s preconditioned conjugate gradient algorithm.

call = function CGPOP-solver(A,x0,b,M⁻¹)
  !*** Compute initial residual ***
  s = Ax0, r = b - s
  rr0 = (GlobalSum(r* r))¹/²

UpdateHalo(r)
  !*** Single pass of regular CG algorithm ***
  z = M⁻¹r, s = z, q = As

UpdateHalo(q)
  {ρ, σ} = GlobalSum{(r * z, s * q)}

!*** Calculate coefficient ***
  α = ρ/σ

!*** Compute next solution and residual ***
  x = x + αs, r = r - αq

do 124 iterations:
  !*** Apply preconditioner ***
  z = M⁻¹r, az = Az

UpdateHalo(az)
  {ρ′, δ, γ} = GlobalSum{(r * z, r * r, az * z)}

!*** Calculate updated coefficients ***
  β = ρ′/ρ, σ = δ - β²ρ, α = ρ/σ, ρ = ρ′

!*** Compute next solution and residual ***
  x = x + α(z + βs)
  r = r - α(az + βq)
  s = z + βs
  q = az + βs

In the 2D MPI variant each process sends data to its four neighbors (north, south, west, and east) by packaging the adjacent data and sending it out. This packaging process is not needed in the 2D CAF variants since the region of the array that needs to be sent out or updated can be addressed directly using ranges with Fortran’s colon operator (for example, the rightmost column of elements in an n by n matrix can be expressed as (n-1, 1:n)).

Buffering also occurs in the 1D MPI variant and some of the 1D CAF variants. Data is aggregated into a buffer so that a single message is sent between a processor and its neighbor. This aggregation is necessary due to the fact that the local data that a neighbor needs may not be stored contiguously in memory. We illustrate the buffering process when using the one-dimensional data structure in Figure 2.3. One of the 1D CAF variants does not use buffering and instead the communication occurs through Coarray reads and/or writes. This is more convenient since the buffering step does not have to be programmed, but it does require multiple messages if the CAF implementation is not able to coalesce communication.

One issue that occurs when using a one-sided communication model, like that in CAF, is that is necessary to decide whether data is pushed so that each processor issues communication calls to putting data to its neighbors, or pulled so that each processor issues get calls to retrieve data from its neighbors. In two-sided communication models both types of communication calls are specified. In order to examine the performance impact of pushing versus pulling data we include variants that push and variants that pull.

A final optimization we examine in the one-dimensional variants is communication/computation overlap. For the one-dimensional variants that use two-sided MPI implementation and CAF, we include versions that include a barrier synchronization step after performing an UpdateHalo and GlobalSum, and versions that overlap communication and computation by only synchronizing between neighboring processes when needed. In Figure 13 we illustrate the tasks that occur for two iterations of the CG algorithm and mark where synchronization is needed. Notice that updating interior data can occur while the boundary exchange is conducted.

In Figures 8, 9, and 11 we show code from the UpdateHalo subroutine in different variants of the miniapp. The most succinct code is in Figure 8, which pulls new values for each ghost cell to update it. Synchronization is necessary on line 17 to ensure that one thread does not modify values in its array while another thread is pulling values from it. Synchronization is necessary on line 7 to ensure that when a thread pulls values from another those values are recent. In Figure 9 we show code that buffers values prior to sending them (as illustrated in Figure 2.3). In Figure 11 we show code that transfers data in the 2D variant of the miniapp.

3. CAF Experimentation

In this section we evaluate the performance of our miniapp implementations on Lynx, a Cray XT5m, and Hopper a Cray XE6. Table 1 provides the specifics of these compute platforms. We com-
piled these implementations using version 7.3.3 of the Cray Fortran compiler. Note that Hopper uses a Gemini interconnect, which provides hardware support for PGAS languages, while the XT5m uses an older SeaStar 2 interconnect, which does not. We illustrate the performance of several of the one-dimensional CGPOP variants in Figure 7. For all of our performance measurements we ran CGPOP so that the conjugate gradient operation was executed 226 times and was applied to a 3600 × 2400 grid of ocean data, therefore Figure 7 present strong scaling results.

The benefit of having hardware support for PGAS can be seen by comparing the execution times on Lynx (dotted lines) and on Hopper (solid lines) shown in Figure 7a. For the one-dimensional MPI version, CGPOP on 768 cores of Hopper is 1.4 times faster than on Lynx. For the CAF unbuffered pull version CGPOP is 14.4 times faster on Hopper versus Lynx. The performance improvement for the CAF unbuffered pull suggests that hardware support has a profound impact on obtainable CAF performance.

On Lynx we observe minor differences in performance between the buffered push, buffered pull, and MPI versions. However, due to not aggregating network traffic, the unbuffered version performs significantly worse than any of the three buffered implementations.

Figure 7. Execution time for several of the one-dimensional implementations of CGPOP on Lynx: a Cray XT5m and Hopper: a Cray XE6.

Figure 12. Code volume difference of several different implementations of CGPOP.
1 subroutine UpdateHalo(array)
2 /* Input parameters and local variables: */
3 real, intent(inout) :: array(:)
4 integer :: i ! dummy counter
5 sync all
6 /
7 / iterate through halo elements, grabbing fresh
8 / values from remote images.
9 /
10 do i=startOfHaloIdx, endOfHaloIdx
11 array(i) = array(halo2grab(i)[haloOnProc(i)]
12 enddo
13 sync all
14 end subroutine UpdateHalo

Figure 8. UpdateHalo subroutine that pulls individual values of
data from neighbors.

The unbuffered version has 44 times slower execution time than the
fastest MPI implementation on 768 cores.

On Hopper just as on Lynx, the performance of the buffered
CAF versions takes no longer than 120% of the execution time of the
MPI version when executed on fewer than 1000 cores. For core
counts greater than 1000 the execution times of the buffered CAF
versions takes no longer than 187% of the time of the MPI version.

As mentioned earlier, the performance penalty for using un-
buffered CAF is much less severe on Hopper versus Lynx. On 768
cores of Hopper the execution time of the unbuffered version is
2.3 times larger than the MPI version. The performance penalty
for use of the unbuffered CAF version grows at larger core counts.
At 28,992 cores the execution time of the unbuffered version takes
16 times as long as the MPI version. Note that on 28,992 cores,
the cost of the CGPOP miniapp is dominated by the cost of the 3-
word GlobalSum, which consumes nearly 80% of the total time.
Fine-grained synchronization, like that needed for the GlobalSum
operation amplify the OS jitter problem [14]. Thus, an efficient and
easy to use CAF reduction operator would significantly benefit the
scalability of the CAF version of CGPOP.

In Figure 7b we compare the execution time of a variant of the
miniapp that includes a computation/communication overlap.
This overlap does not make a significant impact on total execution
time when compared against the buffered push variant. One of the
overlapping variants we test uses a preprocessor directive to instruct
the CAF compiler to not automatically insert synchronization. This
directive is necessary in order to ensure an overlap. We discuss this
issue in more detail in section 4.4.

3.2 Code Volume Comparison

To evaluate the effect CAF has on code volume we use a delta-
SLOC metric. This metric indicates how lines of code change from
a corresponding MPI implementation of CGPOP.

The delta-SLOC metric consists of three components: how
many lines have been modified from the MPI version, how many
lines have been added, and how many lines have been deleted.
Modified lines include some shared text between the base and com-
pared implementation, added lines are not included in the base im-
plementation but are in the compared implementation, and deleted

1 subroutine UpdateHalo(array)
2 /* Input parameters and local variables: */
3 real(r8), intent(inout) :: array(:)
4 integer(i4) :: src, dest, len, ipt, tag
5 integer(i4) :: ierr, i, placeval, j
6 /
7 / Gather data to be sent into a buffer */
8 do i=1, lenSendBuffer
9 sendBuffer(i) = array(halo2send(i))
10 enddo
11 /
12 / Push data from buffer to neighbors */
13 do i=1, nSend
14 ipt = ptrSend(i)
15 len = SendCnt(i)
16 dest = sNeigh(i) + 1
17 placeval = place(i)
18 recvBuffer(placeval:placeval+len-1) = &
19 sendBuffer(ipt:ipt+len-1)
20 enddo
21 sync all
22 /
23 / Indirect address from the recvBuffer to the
24 / receiving side’s array
25 /
26 do i=1, lenRecvBuffer
27 array(recv2halo(i)) = recvBuffer(i)
28 enddo
29 sync all
30 end subroutine UpdateHalo

Figure 9. UpdateHalo subroutine that buffers data prior to pushing it.
/*
 * Iterate through local blocks to send
 * boundary data to neighbors
 */
do i = 1, nLocalBlocks
/*
 * Get information about this local block
 */
glbBlk = local_blocks(i)
call get_block_parameter(&
glbBlk, ib = ib, ie = ie, jb = jb, je = je)
/*
 * If there is a neighboring block to the west
 * send that neighbor data
 */
if (Neigh(west, glbBlk) == 0) then
proc = mDist%proc(Neigh(iwest, glbBlk))
block = mDist%localBlkID(Neigh(iwest, glbBlk))
array(i = i + 1: i + nghost, j = j, block)[proc] = &
array(ib = ib + nghost - 1, jb = je, i)
end if
...
end do

Figure 11. Code from UpdateHalo subroutine that pushes data in
2D CAF variant of miniapp.

lines exist in the base implementation but are nonexistent in the
compared implementation.

We calculate the delta-SLOC metric by using the Unix diff
utility, which reports on added, changed, and deleted lines of text.
We filter source files to exclude whitespace and comments and pass
diff the -d flag to find a minimal set of changes. When the diff
utility reports that n lines in the MPI implementation have been
changed into m lines in the compared implementation and n > m
we consider this as m modified lines and n – m deleted lines. When
n < m we consider this as n modified lines and m – n added lines.

In Figure 12a we present the delta-SLOC measurements of the
CGPOP and POP implementations. We stack the bars for lines-of-
code that are required. The sum of these values is equal to the num-
er of lines of code that differ between each implementation and the base
MPI version. For each version we also include a bar for the num-
ber of lines that were removed from the base MPI version. In Fig-
ure 12b we present this difference subtracted from the number of
deleted lines, which illustrates how the code-volume has changed
from the MPI implementation. In general, a low sum of modified
and added lines indicates that less work would be required to refac-
tor the source application than a high sum. Compared to the buffered
implementations, unbuffered versions require fewer lines of code
to implement and remove more lines of code from the base imple-
mentation. The delta-SLOC metrics are nearly identical between
the buffered pull and push versions.

The two-dimensional variants of the miniapp saw the largest
reduction in code volume due to the fact that explicit buffering was
not necessary in the CAF implementations. The one-dimensional
unbuffered pull variant also saw a large reduction in code volume,
however, the buffered variants did not see the same impact.

Ideally, we could create a variant of the miniapp that had the
reduced code volume benefits of an unbuffered version with the
performance of the buffered version. This may be possible through
future compiler optimization work. For example, because the com-
munication pattern remains constant throughout the lifetime of the
application and the UpdateHalo function is performed once per it-
eration of the solver, a potential optimization could be to inspect
the coarray data access pattern at runtime during the first call to
UpdateHalo and to determine how communication could be aggre-
gated to automatically buffer communication on subsequent calls to
UpdateHalo.

4. Discussion of Experience Using CAF

This section discusses our experience and issues we encountered
while creating CAF variants of CGPOP. Unfortunately, unlike the
experience of others [27] where a significant improvement in ap-
plication performance was achieved when utilizing CAF, we were
unable to achieve any performance advantage from CAF versus the
original MPI implementation. Use of CAF results in a modest to
significant performance penalty. In CGPOP the necessary commu-
nication algorithm is well suited to the existing 2-sided MPI se-
matic, unlike for [27] in which CAF enabled the use of a new and
more efficient communication algorithm.

4.1 Pushing versus pulling

One large difference between programming with MPI and a PGAS
language is that PGAS languages necessarily use one-sided com-
munication. In two-sided communication the sender explicitly in-
vokes a send operation to send data and the receiver explicitly in-
vokes a receive operation to receive data. On the other hand, in
one-sided communication a process may explicitly invoke a get
operation to retrieve data from the local memory of another pro-
cess without having the other process explicitly specify that the
communication should occur. The one-sided put operation enables
a process to place data into the local memory of another process
without the need for the other to specify that such an operation
should occur.

We found implementing the communication of the halo in CG-
POP with a pull (i.e., get) or with a push (i.e., put) equal in terms
of their impact on code volume (see Figure 12). We expect that push
variants of CGPOP will perform faster than pull variants because
pulling requires that the communication routine use two messages
to conduct data transfer. One of these messages is to inform the
target process that a get operation is invoked, and the other mes-
sage sends the requested data. Despite the necessity of this extra
message we found the performance improvement of pushing over
pulling is minimal (see Figure 7).

4.2 Need for data aggregation

During the boundary exchange routine for the 1D versions of the
miniapp data the halo may not be contiguously stored in memory.
In the unbuffered variant of the code (see Figure 8) we conduct this
transfer by scanning through the local border points and invoking
individual get operations for each point. However, this approach
introduces a huge performance penalty due to the fact that each
individual get operation generates a unique message and thus
introduces a large amount of per-message overhead.

To get around this we first gather data into a buffer so that all
data that is to be pushed out to a neighbor or pulled from a neigh-
bor is contiguous. This alleviates the performance issues, but intro-
duces the need for explicit marshalling. Ideally, there could be some
mechanism to decouple an individual put or get command from an
individual message. In [6] Chen et al. use an automatic communica-
tion coalescing optimization in order to improve performance.

One feature that would improve would be for CAF to include a
statement to send non-contiguous data and automatically aggregate
it for communication. For example, with compiler support, lines 15
through 32 in Figure 9 could be replaced by the simple expression:

array(recv2halo(:)[dest(i)])[dest(i)] = &
array(halo2send(:))

4.3 Distribution of communication metadata

One interesting difficulty we encountered while writing the CAF
version of the communication routine is that we had to modify the
distribution of metadata. This issue arises due to the different in-
formation necessary to conduct a data-transfer for one-sided ver-
sus two-sided communication. CAF is a one-sided communication
model, which means that only one side is required to specify that
communication should occur: either to get or put some piece of data
on another image.

Regardless of whether put or get operations are used to com-
municate, the side specifying the communication requires the im-
age number of the other side, as well as the address of where
data should be pulled from or the address of where data should
be placed. With two-sided communication only the sending side
needs to be aware of the address to pull data from and only the
receiving side needs to be aware of the address to place data. In
the MPI scheduling object for the MPI implementation, metadata
specifying what communication occurs was distributed to work for
two-sided communication (information about where to place data
was stored on the receiving side). Thus it was necessary to modify
the distribution of this metadata so that either push or pull one-sided
communication could occur.

4.4 Synchronization Management

In order to enable a communication/computation overlap in CAF
with version 7.3.3 of the Cray compiler we found it necessary to use the defer_sync compiler directive. Without this directive
the Cray compiler conservatively forces synchronization in order
to ensure program semantics. In the case of our communication
overlap code the compiler would insert a synchronization before
the start communication routine returned, effectively eliminating
the ability to overlap. We were able to determine that this was the
case by examining the assembly output provided by the compiler.

4.5 Missing Reductions on Some Machines

Although we used Coarray Fortran for the point-to-point commu-
nication done in the boundary exchange routine, we found it nec-
tessary to use MPI for collective communication done elsewhere
in the program. Broadcast calls are used to propagate global para-
eters, and a sum-reduction operation is used in the conjugate-
gradient algorithm itself. Although the Fortran standard does not
include broadcast or reduction operations for Coarrays, as of ver-
sion 7.3.3 Cray’s compiler does: namely, through its CO_SUM and
CO_BCAST routines. Unfortunately, these routines are currently
only supported on Cray XE machines.

4.6 Synchronization Overhead

One impact on the performance for Coarray version was the over-
head necessary to synchronize computation and data. In Figure 13
we illustrate the tasks needed to complete two update steps and the
dependencies that exist between the tasks. The dependences drawn
using arrows with hollow heads represent a dependence that exist
across images (aka processes) and thus require a synchronization
step using either a ‘sync all’ or ‘sync images’ statement. Since these
cross images dependences exist between a block and all of neigh-
bors it is possible to pass the sync images statement and pass it
to a team of processes that includes the process itself and its neigh-
bors. Doing so improves performance over using sync all. In the
overlapped variants of the miniapp we use sync images with such a
team.

One interesting issue we have discovered is that on a Cray
XT6m the sync all statements are more costly than MPI_Barrier.
In this paper we examine how Coarray Fortran compares to MPI by comparing implementations of the CGPOP miniapp against each other. How well parallel programming models improve performance and programmer productivity is typically evaluated within the context of smaller kernels. Some examples include sparse-matrix vector multiply [18], Smith-Waterman [11], and dense matrix computations and FFT [25]. Although such kernels play an important role in evaluating programming models and computer platforms, their scope is limited and they do not necessarily have the same performance profiles as larger applications. Heroux et al. introduced the process of defining a miniapp as a research tool that can encapsulate the characteristics of a full application [17], which is the approach we use in this paper.

In some contexts the NAS Parallel Benchmarks [3] may be considered miniapps, and they are commonly used to evaluate programming models. In [7] Datta et al. use NPB to evaluate Titanium; and in [4] Cantonnet et al. compare UPC [12] to MPI using the NPB CG benchmark. In [23] Malli et al. compare the performance of MPI versus OpenMP versus UPC implementations of NPB on multicore systems. Although NAS includes a conjugate-gradient benchmark, this benchmark does not fulfill the requirement of being a performance proxy of POP. We illustrate the performance of POP and NPB in Figure 15. The scaling properties of CG significantly depend on the underlying mesh, or graph that represents the sparse matrix nonzero structure [15, 16] and NPB, unlike POP, uses a random pattern of nonzeros in the sparse matrix.

Cantonnet et al. [4] compared UPC [12] to MPI in terms of programmability within the context of the NPB CG benchmark. In addition to SLOC, they use conceptual complexity metrics such as the number of keywords, function calls, and parameters to measure programmability. Such complexity metrics are complimentary to the ones we use in our comparison and could be added to a miniapp-based programming model evaluation.

Another set of benchmarks is the DARPA HPC Challenge (HPCC) Suite. [22] From 2005 to 2010 the HPCC suite has been used in the HPC Challenge Competition, where participants enter implementations of the benchmarks in various languages to compete for the “most productive” implementation. In previous years, entries written in PGAS and HPCS languages such Chapel [5], CAF 2.0 [24], UPC, and X10 [10] have been involved in this competition.

There has also been some work where full applications have been completely rewritten in a new parallel programming model to enable a performance and programmability evaluation [29, 31]. In [31] Yelick et al. also mention the importance of optimizing for fine grained access.

In [30] Coarrays are integrated into an older version of POP and tested on the Cray X1. The advantage of the CAF implementation in [30] were partially a result of reducing communication volume. The reduction in communication volume was an algorithm improvement and not a function of program languages. A similar reduction in communication volume was implemented in the MPI version of POP [9] used in this study.

6. Conclusions

In this paper we evaluate Coarray Fortran using the CGPOP miniapp. Currently, we were unable to see a performance benefit from using CAF over MPI on this code. Code volume was improved in the two-dimensional data structure variants of miniapp when using CAF because in array assignment expressions borders can be expressed using Fortran’s array slicing features. However, in the one-dimensional data structure version a buffering step was needed for performance, which negatively impacted code volume. As CAF matures we expect that the performance of CGPOP will improve. CAF could benefit from automatic communication coalescing and more efficient synchronization mechanisms. Additionally, we believe CAF could also be improved by adding language features that aid with scatter/gather operations across cores. Also, Cray’s implementation of Fortran could be improved by enabling Coarray reductions to work on additional systems.

Acknowledgements

We thank the reviewers for their helpful comments and suggestions including the idea to include a scatter operator in CAF. This work was supported by Department of Energy Early Career Award #DE-SC003956. This work was financially supported through National Science Foundation Cooperative Grant NSF01, which funds the National Center for Atmospheric Research (NCAR), and through the grant: #OCI-0749206.

References


