OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers

Alessandro Fanfarillo[*]
University of Rome
Tor Vergata
Rome, Italy

Tobias Burnus[†]
Munich, Germany

Valeria Cardellini[‡]
University of Rome
Tor Vergata
Rome, Italy

Salvatore Filippone[§]
University of Rome
Tor Vergata
Rome, Italy

Dan Nagle[¶]
National Center for
Atmospheric Research
Boulder, Colorado

Damian Rouson[II]
Sourcery Inc.
Oakland, California

ABSTRACT

Coarray Fortran is a set of features of the Fortran 2008 standard that make Fortran a PGAS parallel programming language. Two commercial compilers currently support coarrays: Cray and Intel. Here we present two coarray transport layers provided by the new OpenCoarrays project: one library based on MPI and the other on GASNet. We link the GNU Fortran (GFortran) compiler to either of the two OpenCoarrays implementations and present performance comparisons between executables produced by GFortran and the Cray and Intel compilers. The comparison includes synthetic benchmarks, application prototypes, and an application kernel. In our tests, Intel outperforms GFortran only on intra-node small transfers (in particular, scalars). GFortran outperforms Intel on intra-node array transfers and in all settings that require inter-node transfers. The Cray comparisons are mixed, with either GFortran or Cray being faster depending on the chosen hardware platform, network, and transport layer.

Keywords

Fortran, PGAS, Coarrays, GCC, HPC

[*]fanfarillo@ing.uniroma2.it - To whom correspondence should be addressed

[†]burnus@net-b.de

[‡]cardellini@ing.uniroma2.it

[§]salvatore.filippone@uniroma2.it

[¶]dnagle@ucar.edu

[II]damian@sourceryinstitute.org

1. INTRODUCTION

Coarray Fortran (also known as CAF) originated as a syntactic extension of Fortran 95 proposed in the early 1990s by Robert Numrich and John Reid [10] and eventually became part of the Fortran 2008 standard published in 2010 (ISO/IEC 1539-1:2010) [11]. The main goal of coarrays is to allow Fortran users to create parallel programs without the burden of explicitly invoking communication functions or directives such as those in the Message Passing Interface (MPI) [12] and OpenMP [2].

Coarrays are based on the Partitioned Global Address Space (PGAS) parallel programming model, which attempts to combine the SPMD approach used in distributed memory systems with the semantics of shared memory systems. In the PGAS model, every process has its own memory address space but it can access a portion of the address space on other processes. The coarray syntax adds a few specific keywords and leaves the Fortran user free to use the regular array syntax within parallel programs.

Coarrays were first implemented in the Cray Fortran compiler, and the Cray implementation is considered the most mature, reliable, and high-performing. Since the inclusion of coarrays in the Fortran standard, the number of compilers implementing them has increased: the Intel and g95 [15] Fortran compilers as well as compiler projects at Rice University [6] and the University of Houston (OpenUH) [5] support coarrays.

The Cray compiler only runs on proprietary architectures. The Intel compiler is only available for Linux and Windows, and the standard Intel license only supports coarrays in shared memory; running in distributed memory requires the Intel Cluster Toolkit. These technical limitations, in conjunction with the cost of a commercial compiler, limit the widespread usage of coarrays.

The availability of coarray support in a free compiler could contribute to wider evaluation and adoption of the coarray parallel programming model. The free, released compilers

with coarray support, however, do not support several other standard Fortran features, which limits their utility in modern Fortran projects. For example, neither the Rice compiler nor OpenUH nor g95 supports the object-oriented programming (OOP) features that entered the language in the Fortran 2003 standard. Furthermore, g95's coarray support is only free in its shared-memory configuration. Configuring the compiler for multi-node coarray execution requires purchasing a commercial license.

Considering the compiler landscape, the advent of support for coarrays in the free, open-source, and widely used GNU Fortran compiler (GFortran) potentially represents a watershed moment. The most recent GFortran release (4.9.1) supports most features of the recent Fortran standard, including the aforementioned OOP features [4]. Furthermore, the pre-release development trunk of GFortran offers nearly complete support for the coarray features of Fortran 2008 plus several anticipated new coarray features expected to appear in the next Fortran standard. However, any programs that are not embarrassingly parallel require linking GFortran-compiled object files with a parallel communication library. In this paper, we present performance results for OpenCoarrays, the first collection of open-source transport layers that support coarray Fortran compilers by translating a compiler's communication and synchronization requests into calls to one of several communication libraries. We provide a link to the OpenCoarrays source code repository on the dedicated domain http://opencoarrays.org.

Section 2 introduces the coarray programming model. Section 3 introduces the OpenCoarrays transport layers: one based on MPI and one on the GASNet communication library for PGAS languages [1]. Section 4.1 presents the test suite that forms our basis for comparisons between each compiler/library combination. Section 5 presents the bandwidth, latency, and execution times for various message sizes and problem sizes produced with each compiler/library combination. Section 6 summarizes our conclusions.

2. INTRODUCTION TO COARRAYS

In this section, we explain basic coarray concepts. A program that uses coarrays is treated as if it were replicated at the start of execution. Each replication is called an image. Each image executes asynchronously until the programmer explicitly synchronizes via one of several mechanisms. A typical synchronization statement is `sync all`, which functions as a barrier at which all images wait until every image reaches the barrier. A piece of code contained between synchronization points is called a *segment* and a compiler is free to apply all its optimizations inside a segment.

An image has an image index that is a number between one and the number of images (inclusive). In order to identify a specific image at runtime or the total number of images, Fortran provides the `this_image()` and `num_images()` functions. A coarray can be a scalar or array, static or dynamic, and of intrinsic or derived type. A program accesses a coarray object on a remote image using square brackets [ ].

An object with no square brackets is considered local. A simple coarray Fortran program follows:

```
real, dimension(10), codimension[*] :: x, y
integer :: num_img, me

num_img = num_images()
me = this_image()

! Some code here
x(2) = x(3)[7] ! get value from image 7
x(6)[4] = x(1) ! put value on image 4
x(:)[2] = y(:) ! put array on image 2

sync all

! Remote-to-remote array transfer
if (me == 1) then
  y(:)[num_img] = x(:)[4]
  sync images(num_img)
elseif (me == num_img) then
  sync images([1])
end if

x(1:10:2) = y(1:10:2)[4] ! strided get from 4
```

In this example, `x` and `y` are coarray arrays and every image can access these variables on every other image. All the usual Fortran array syntax rules are valid and applicable. Also, a coarray may be of user-defined derived type.

Fortran provides Locks, Critical sections and Atomic Intrinsics for coarrays. Currently, the Cray compiler is the only released compiler supporting these features. Although not discussed in this paper, the OpenCoarray and GNU Fortran trunks offer partial support for these features.

3. GNU FORTRAN AND LIBCAF

GNU Fortran (GFortran) is a free, efficient and widely used compiler. Starting in 2012, GFortran supported the coarray syntax for single-image execution, but it did not provide multi-image support. GFortran delegates communication and synchronization to an external library (LIBCAF) while the compiler remains agnostic with regards to the actual implementation of the library calls.

For single-image execution, GFortran calls stub implementations inside a LIBCAF_SINGLE library included with the compiler. For multi-image execution, an external library must be provided to handle GFortran-generated procedure invocations. Having an external library allows for exchanging communication libraries without modifying the compiler code. It also facilitates integration of the library into compilers other than GFortran so long as those compilers are capable of generating the appropriate library calls. OpenCoarrays uses a BSD-style, open-source license.

OpenCoarrays currently provides two alternative versions of the library: LIBCAF_MPI and LIBCAF_GASNet. The MPI version has the widest features coverage; it is intended as the default because of the ease of usage and installation. The GASNet version targets expert users; it provides better performance than the MPI version but requires more effort during the installation, configuration, and usage.

3.1 LIBCAF_MPI

LIBCAF_MPI currently supports

- Coarray scalar and array transfers, including efficient

transfers of array sections with non-unit stride.

- Synchronization via `sync all`, `sync images`.
- Collective procedures (`co_sum`, `co_max`, `co_min`, etc...) except for character coarrays.
- Atomics.

The support for coarray transfers includes get/put and strided get/put for every data type, intrinsic or derived.

LIBCAF_MPI uses the one-sided communication functions provided by MPI-2 and MPI-3 with passive synchronization (using MPI_Win_lock/unlock).

This apporach allows us to easily implement the basic coarrays operations and to exploit the MPI RMA support when available.

3.1.1 Get, Put and Strided Transfers

A Get operation consists of a transfer from a remote image to a local image; in coarray syntax it is expressed with the local data on the left-hand side of the equality and the remote data on the right-hand side. It maps directly onto the MPI_Get one-sided function introduced in MPI-2. A Put operation consists in the opposite of a Get operation; it transfers data from the local image to a remote image. It is expressed in corray syntax with the local data on the right hand side of the equality and the remote data on the left hand side. This operation maps onto the MPI_Put one-sided function introduced in MPI-2. For strided transfer, we mean a non-contiguous array transfer. This pattern is pretty common in scientific applications and usually involves arrays with several dimensions. LIBCAF_MPI provides two approaches for the strided transfer implementation: sending element-by-element and using the Derived Data Types feature provided by MPI. The first approach is the most general, easy to implement and inefficient in terms of performance. It is provided as default configuration because of its generality. The second approach provides higher performance but requires more memory in order to transfer the data.

3.2 LIBCAF_GASNet

GASNet stands for Global Address Space Networking and is provided by UC Berkeley [1]. LIBCAF_GASNet is an experimental version targeting expert users but providing higher performance than LIBCAF_MPI. GASNet provides efficient remote memory access operations, native network communication interfaces and useful features like Active Messages and Strided Transfers (still under development). The major limitation of GASNet for a coarray implementation consists in the explicit declaration of the total amount of remote memory required by the program. Thus, the user has to know, before launching the program, how much memory is required for coarrays. Since a coarray can be static or dynamic, a good estimation of such amount of memory may not be easy to guess. A memory underestimation may generate sudden errors due to memory overflow and overestimations may require the usage of more compute nodes than needed.

Currently, LIBCAF_GASNet supports only coarray scalar and array transfers (including efficient strided transfers) and all the synchronization routines. This paper therefore provides only a partial analysis of this version. We plan to complete every test case in future work.

LIBCAF_GASNet maps the Get and Put operations directly onto the gasnet_put_bulk and gasnet_get_bulk functions. As LIBCAF_MPI, LIBCAF_GASNet offers two alternatives for the strided transfers: the element-by-element approach and the native support for strided transfers provided by GASNet. The second approach uses the gasnet_putv_bulk and gasnet_getv_bulk functions. They provide good performance even if they are not yet optimized.

4. COARRAY COMPARISON

This section presents a comparison between the GFortran/OpenCoarrays runtime performance and that of the Cray and Intel commercial compilers. We analyze LIBCAF_MPI more deeply but also provide some LIBCAF_GASNet results on Cray machines.

4.1 Test Suite

In order to compare LIBCAF with the other compilers, we ran several test cases.

- EPCC CAF Micro-benchmark suite.
- Burgers Solver.
- CAF Himeno.
- Distributed Transpose.

We describe each of these test codes in more detail next.

4.1.1 EPCC CAF Micro-benchmark Suite

David Henty of the University of Edinburgh wrote the EPCC CAF Micro-benchmark suite [8]. Its source code is freely available on the web. It measures the performance (latency and bandwidth) of the basic coarray operations (get, put, strided get, strided put, sync), a typical communication pattern (the halo exchange), and synchronization. Every basic operation is analyzed in two different scenarios: single point-to-point and multiple point-to-point. In the first case, image 1 interacts only with image n; every other image waits for the end of the test. In this scenario, no network contention occurs; it thus represents a best case scenario.

During the multiple point-to-point tests, image i interacts only with image $i+n/2$; this test case models what actually happens in real parallel applications. Here we compare only these two scenarios. We do not report results on the performance of synchronization operations and halo exchanges. The synchronization time has a smaller impact on the overall performance than the transfer time, and the halo-exchange performance is considered in the real application tests like CAF Himeno.

4.1.2 Burgers Solver

Rouson et al. [14] wrote the coarray Fortran Burgers Solver, and the source code is freely available online.[1] This application prototype solves a partial differential equation (PDE)

[1]See "Resources"' at http://www.cambridge.org/Rouson

that involves diffusion, nonlinear advection, and time dependence. It uses Runge-Kutta time advancement. It also uses halo point exchanges to support finite difference approximations. The spatial domain is one-dimensional, which keeps the code much simpler than most scientific PDE applications. Furthermore, in studies with the Cray compiler, the Burgers solver exhibits 87% parallel efficiency in weak scaling on 16,384 cores. The solver exhibits linear scaling (sometimes super-linear) [7] and a separate, upcoming paper provides comparisons to a MPI version of the solver. It uses coarrays mainly for scalar transfers. The destination images of transfers are neighboring images that are usually placed on the same node.

4.1.3 CAF Himeno

Ryutaro Himeno of RIKEN wrote the initial version of CAF Himeno as a parallel application using OpenMP and MPI to build a three-dimensional (3D) Poisson relaxation via the Jacobi method. William Long of Cray, Inc., developed the first coarray version of CAF Himeno using an early Cray coarray implementation. Dan Nagle of NCAR refactored the coarray CAF Himeno to conform to the coarray feature set in the Fortran 2008 standard. The resulting test expresses strided array transfers in the usual Fortran colon syntax.

4.1.4 3D Distributed Transpose

Robert Rogallo, formerly of NASA Ames Research Center, provided the Distributed Transpose test. This application kernel was extracted from codes for 3D Fourier-spectral, direct numerical simulations of turbulent flow [13]. The kernel is available in coarray and MPI versions, facilitating comparison between the two.

4.2 Hardware and Software

In order to compare the Cray and Intel coarray implementations with our new GFortran/OpenCoarray implementation, we ran each test case on high-performance computing (HPC) clusters provided by several organizations. Our Intel CAF tests utilize the Intel Cluster Toolkit as required for distributed-memory coarray execution. Or Cray compiler tests use proprietary Cray hardware as required. Our GFortran/OpenCoarray tests based on MPI (LIBCAF_MPI) can be executed on any machine able to compile gcc and any standard MPI implementation. The hardware available for our analysis is as follows:

- Eurora: Linux Cluster, 16 cores per node, Infiniband QDR 4x QLogic (CINECA).
- PLX: IBM Dataplex, 12 cores per node, Infiniband QDR 4x QLogic (CINECA).
- Yellowstone/Caldera[2]: IBM Dataplex, 16 cores per node (2 GB/core), FDR Infiniband Mellanox 13.6 GBps (NCAR).
- Janus: Dell, 12 cores per node, Infiniband Mellanox (CU-Boulder).
- Hopper: Cray XE6, 24 cores per node (1.3 GB/core), 3-D Torus Cray Gemini (NERSC).
- Edison: Cray XC30, 24 cores per node, Dragonfly Cray Aries (NERSC).

[2]For tests requiring more than 2GB/core we used Caldera, which has 4 GB/core and the same processors as Yellowstone.

In this paper we present only the results collected from Yellowstone and Hopper/Edison. In particular, the comparison between GFortran and Intel has been run on Yellowstone and the comparison between GFortran and Cray on Hopper/Edison. In order to validate the results, we ran the tests on the remaining machines listed above.

4.2.1 Hopper and Edison

Hopper allows for running coarray programs only with Cray and GFortran. For the Cray compiler we used the 8.2.1 version with the -O3 flag, loaded the craype-hugepages2M module and set the XT_SYMMETRIC_HEAP_SIZE environment variable properly (when needed). For GFortran we used the GCC 5.0 development version (the GCC trunk) and Mpich/7.0.0 (installed on the machine) for LIBCAF_MPI. The only flag applied on GFortran was -Ofast (for optimization). The GCC 5.0 experimental version employed was almost the same version present as of this writing on the gcc-trunk (no performance changes).

We used Edison only for the EPCC CAF micro-benchmark. On Edison, we used the Cray compiler version 8.3.0 with the same configuration set on Hopper.

4.2.2 On Yellowstone

Yellowstone allows us to run only coarray programs compiled with Intel and GFortran. We used the Intel compiler 14.0.2, which employs IntelMPI 4.0.3 for the coarrays support. We applied the following flags the compilation: -Ofast -coarray -switch no_launch. For GFortran, we used the GCC 5.0 development version (same used for Cray) and MPICH IBM (optimized for Mellanox IB) for LIBCAF_MPI. Even in this case, the only flag applied on GFortran was -Ofast.

5. RESULTS

In this section we present an exhaustive set of tests performed on single and multiple nodes of Yellowstone/Caldera, Hopper, and Edison. "Single node"' means that the network connecting the cluster nodes is not involved; such a configuration can be useful to understand the performance of the communication libraries in a shared-memory environment.

5.1 EPCC CAF - GFortran vs. Intel

The EPCC CAF micro-benchmark suite provides latency and bandwidth for several block sizes during the basic CAF operations (get, put, strided get, strided put). EPCC tests two scenarios: single point-to-point, where there is no network contention involved, and multiple point-to-point for a more realistic test case. In this section, we present the results of GFortran vs. Intel on single and multiple nodes only for the put and strided put operations ("get" has almost the same results). This particular comparison, GFortran vs. Intel, is feasible only on Yellowstone/Caldera.

5.1.1 Single pt2pt Put on a single node

This test employed 16 cores in one point-to-point put operation on one 16-core node on Yellowstone. Figures 1 and 2 show that, on a single node, Intel is better than GFortran for quantities less than or equal to 4 doubles (32 bytes total). After that point the latency assumes an exponential trend for Intel but stays constant for GFortran. The bandwidth, after 4 doubles, has exactly an inverse behavior: exponential for GFortran and constant for Intel. In other words, for small transfers (specifically scalars) within the same node, without contention, Intel outperforms GFortran.

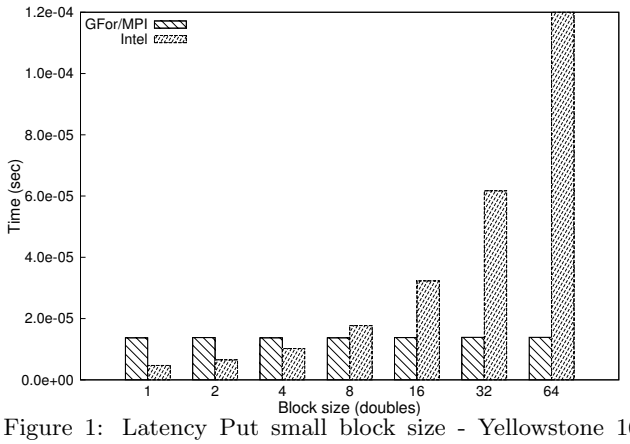

Figure 1: Latency Put small block size - Yellowstone 16 cores

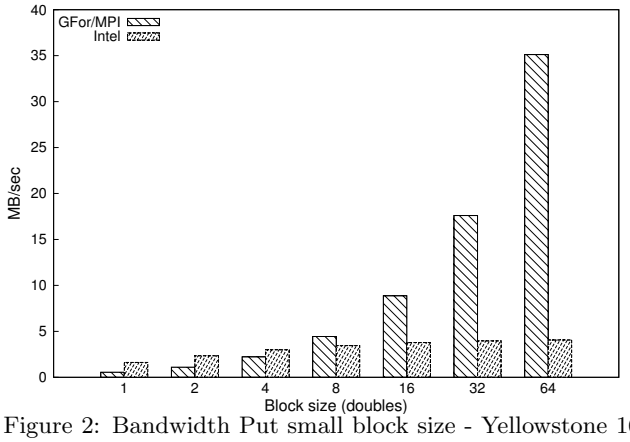

Figure 2: Bandwidth Put small block size - Yellowstone 16 cores

Figures 3 and 4 show that increasing the block size does not change the trends observed on small block sizes.

5.1.2 Multi-pt2pt Put on a single node

This test employed 16 cores in multiple point-to-point put operations on one 16-core node on Yellowstone. In this configuration, image i interacts only with image $i + n/2$ (where n is the total number of images). For this case, we show only the bandwidth in Figure 5.

This test case shows that Intel is less affected by network contention (in this case the shared memory network) than GFortran. In this particular case, Intel has a bigger bandwidth than GFortran for values less than or equal to 8 doubles (64 bytes).

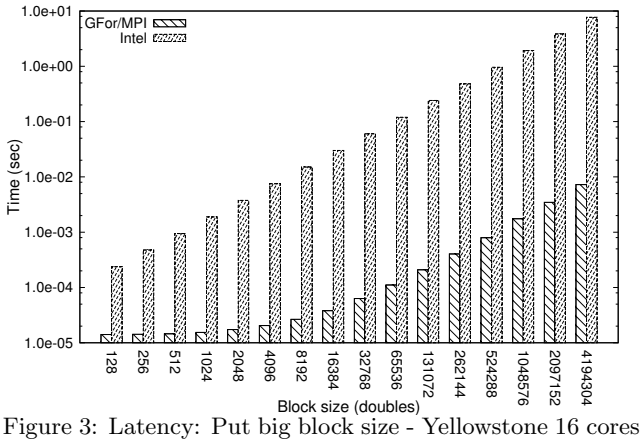

Figure 3: Latency: Put big block size - Yellowstone 16 cores

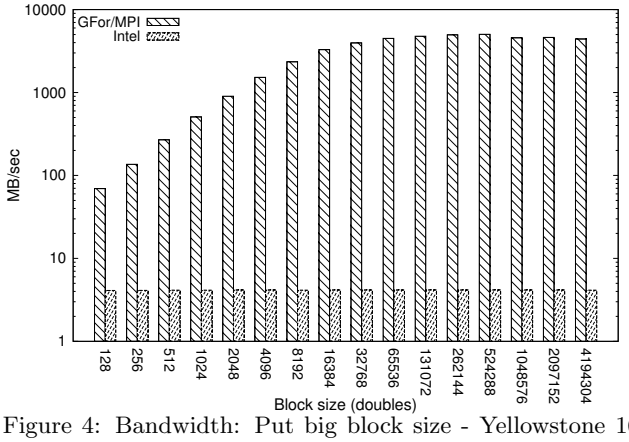

Figure 4: Bandwidth: Put big block size - Yellowstone 16 cores

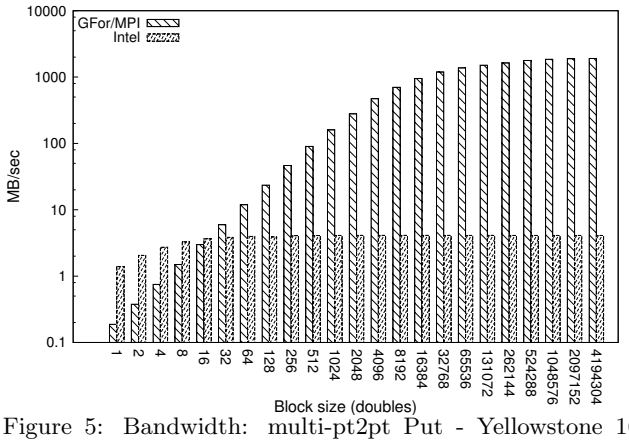

Figure 5: Bandwidth: multi-pt2pt Put - Yellowstone 16 cores

A good way to see this phenomenon is to chart the bandwidth difference between single and multiple. Figure 6 shows such a comparison, and we can see that Intel has a constant behavior. Intel is insensitive to network contention, which means the network is not the bottleneck for Intel. GFortran exhibits a quite different behavior, implying that GFortran is network-limited.

5.1.3 Single pt2pt Strided Put on a single node

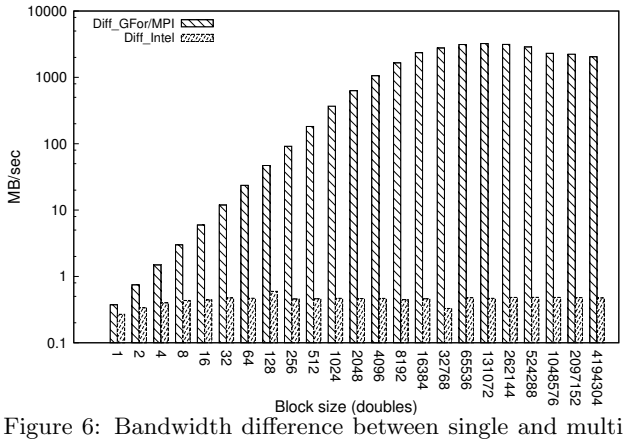

Figure 6: Bandwidth difference between single and multi - Yellowstone 16 cores

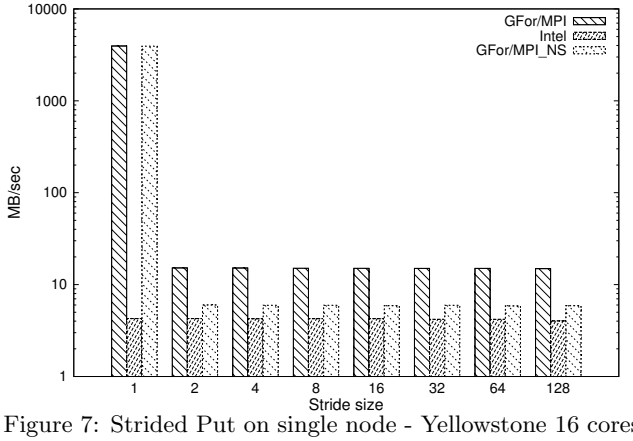

Figure 7: Strided Put on single node - Yellowstone 16 cores

By "strided transfer," we refer to a non-contiguous array transfer. This kind of transfer is common in several scientific applications, and its performance is therefore crucial for the performance of the entire application. That test is also very useful to understand the behavior of Intel. LIBCAF_MPI supports efficient strided transfer; such support can be disabled by sending the array elements one at time. Figure 7 shows the performance during the strided transfer of LIBCAF_MPI and Intel; GFor/MPI_NS represents the performance of LIBCAF_MPI without strided transfer support (sending element-by-element). The most interesting fact is that, even with a contiguous array (stride = 1), Intel has the same performance of LIBCAF_MPI without the strided transfer support, indicating that Intel sends arrays element-by-element even with contiguous arrays.

LIBCAF_MPI uses the classic MPI Data Type in order to implement an efficient strided transfer. This approach is very easy to implement but it is not very efficient either in terms of memory or time.

5.1.4 Single pt2pt Put on multiple nodes

In this configuration, we ran the benchmark on 32 cores, thus involving the real, inter-node network in the transfer. Something unexpected happens: in Figure 8, Intel shows about 428 seconds of latency for transferring 512 doubles (4 KBytes) through the network. This strange behavior relates to element-wise transfer.

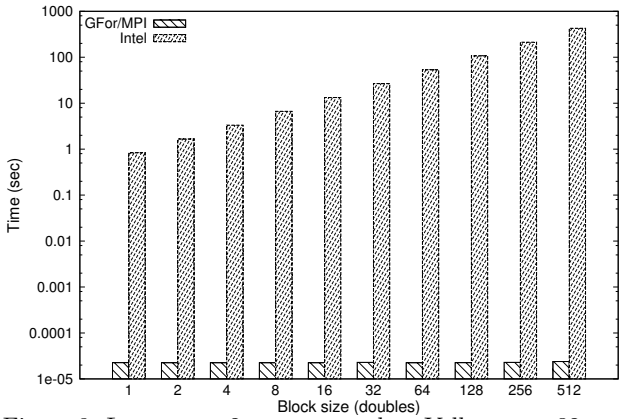

Figure 8: Latency on 2 compute nodes - Yellowstone 32 cores

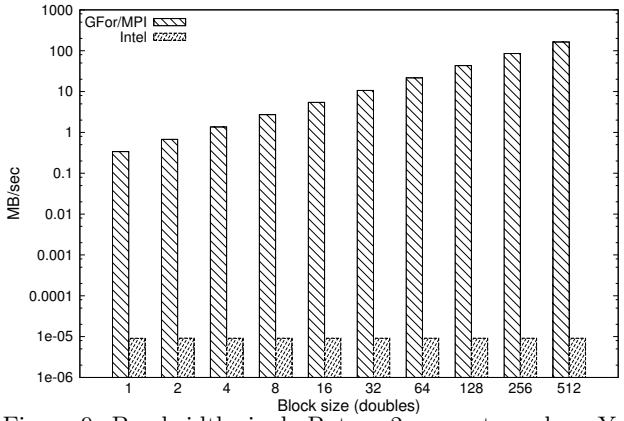

Figure 9: Bandwidth single Put on 2 compute nodes - Yellowstone 32 cores

5.2 EPCC CAF – GFortran vs. Cray

In this section, we compare the results of GFortran and Cray. This particular comparison is feasible only on Hopper and Edison. For this case, we report only the bandwidth results, and we report the GASNet results for EPCC and the Distributed Transpose tests. Furthermore, we report only the results for the Get and Strided Get operations.[3]

5.2.1 Single pt2pt Get on a single node

This test involves one point-to-point Get on 24 cores. Because Hopper and Edison have 24 cores on each compute node, the test runs on one node. Figure 10 shows that, for small transfers, LIBCAF_GASNet outperforms Cray on Hopper. For big transfers, Figure 11 shows that Cray is usually (but not always) better than the two LIBCAF implementations. Figures 12 shows that, on Edison, LIBCAF_GASNet outperforms Cray for small transfers (like on Hopper) but with a bigger gap. Figure 13 shows that, for big sizes, LIBCAF_MPI outperforms Cray and LIBCAF_GASNet.

On single node, on Edison, Cray is always outperformed by one of the LIBCAF implementations.

[3]The performance of Get is similar to Put, but we do not show the latter results because of RMA problems when data packages exceed a protocol-type-switching threshold.

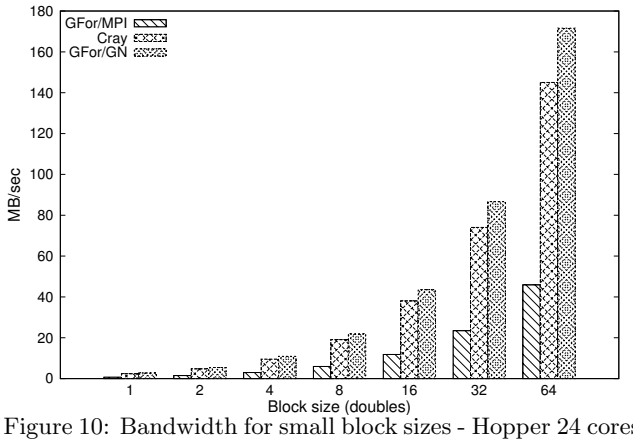

Figure 10: Bandwidth for small block sizes - Hopper 24 cores

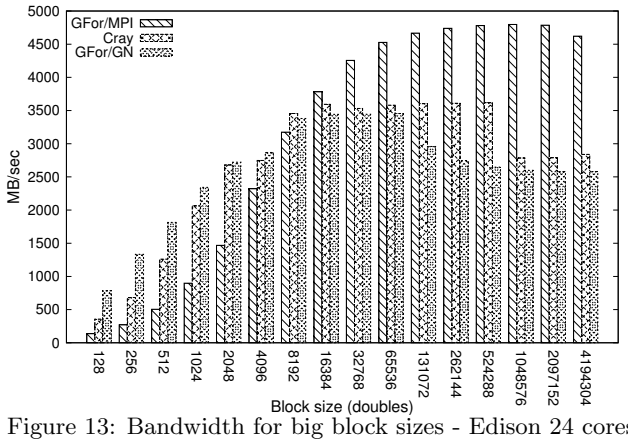

Figure 13: Bandwidth for big block sizes - Edison 24 cores

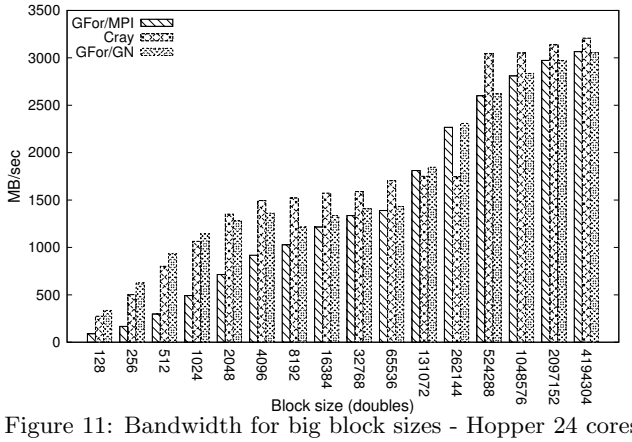

Figure 11: Bandwidth for big block sizes - Hopper 24 cores

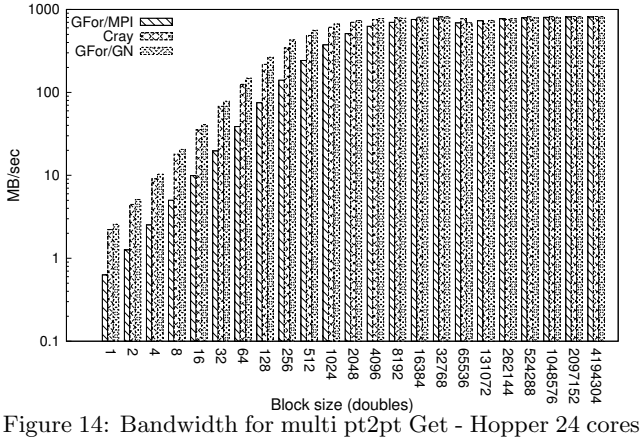

Figure 14: Bandwidth for multi pt2pt Get - Hopper 24 cores

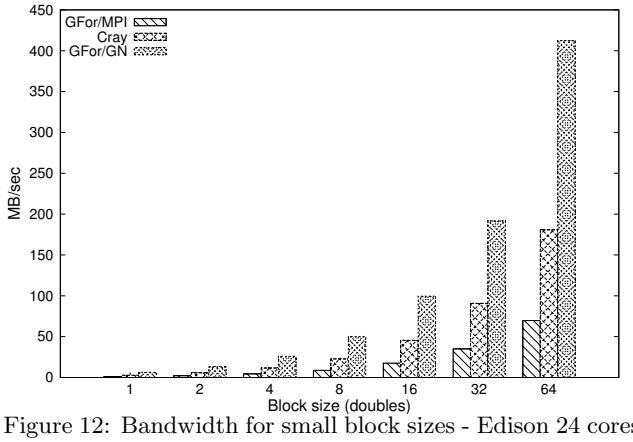

Figure 12: Bandwidth for small block sizes - Edison 24 cores

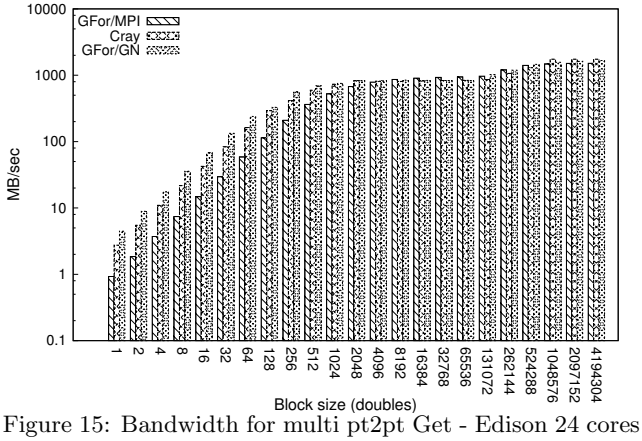

Figure 15: Bandwidth for multi pt2pt Get - Edison 24 cores

5.2.2 *Multi pt2pt Get on a single node*

Multiple point-to-point Get on 24 cores. We analyze the behavior of LIBCAF_MPI, LIBCAF_GASNet and Cray with contention on the underlying network layer. Figure 14 shows LIBCAF_GASNet outperforming Cray in almost every case for which there exists contention on Hopper's underlying network layer. Figure 15 shows LIBCAF_GASNet exhibiting better performance than Cray for small block sizes on Edison.

5.2.3 *Single pt2pt Strided Get on a single node*

Figure 16 shows the performance of one point-to-point strided Get on one Hopper compute node. In this case, Cray always has the best performance. GASNet provides experimental strided-transfer support that is not yet optimized. The single stride has dimension 32768 doubles. Figure 17 shows that LIBCAF_MPI is very effective on Edison. In fact, for several stride sizes, LIBCAF_MPI is better than Cray.

5.2.4 *Single pt2pt Get on multiple nodes*

Figures 18 and 19 show that on Hopper, on multiple nodes, Cray performs better than LIBCAF in almost every situa-

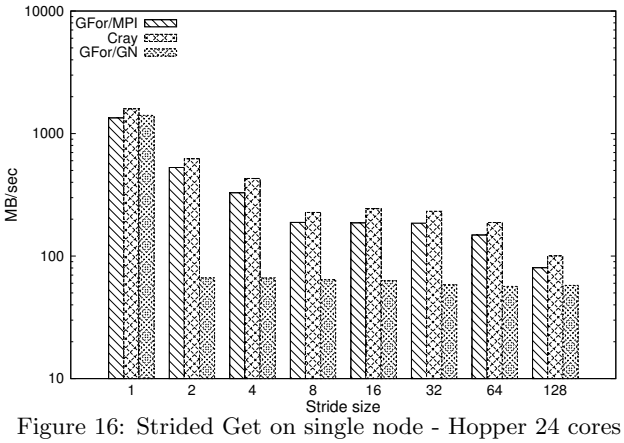

Figure 16: Strided Get on single node - Hopper 24 cores

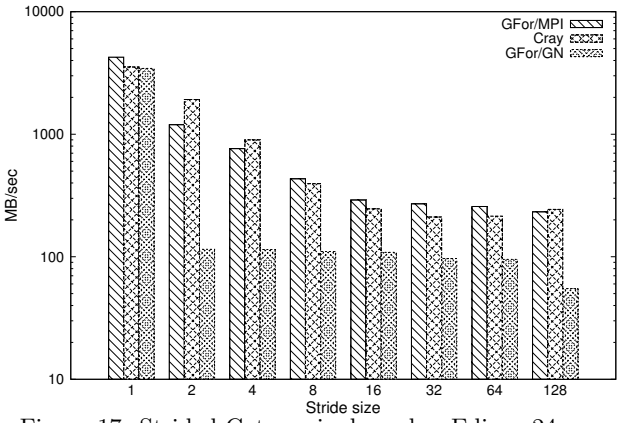

Figure 17: Strided Get on single node - Edison 24 cores

tion. GASNet shows good behavior for small block sizes. Figures 20 and 21 show that, on Edison, LIBCAF_GASNet almost always outperforms Cray.

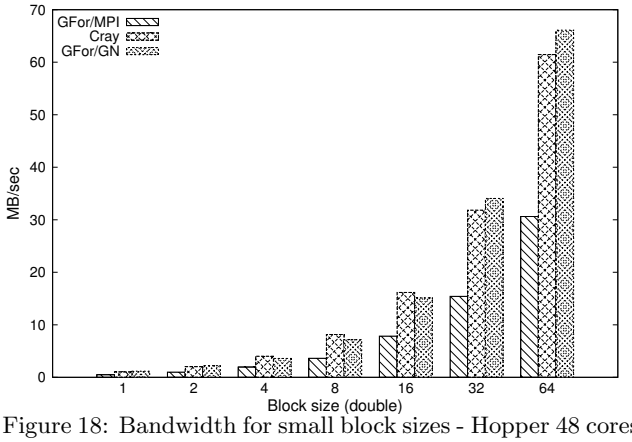

Figure 18: Bandwidth for small block sizes - Hopper 48 cores

5.2.5 *Multi pt2pt Get on multiple nodes*

Figures 22 and 23 show the performance of Cray and LIBCAF when network contention is involved on multiple nodes. On Hopper, LIBCAF_GASNet has the best performance for small transfers (less than 512 doubles); on Edison, LIBCAF_GASNet shows good performance for transfers smaller than 16 doubles.

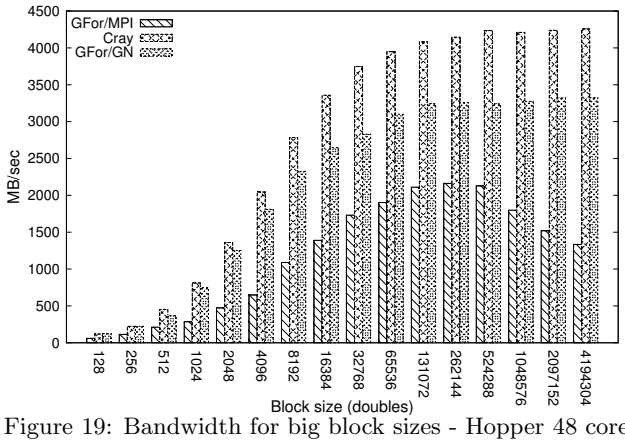

Figure 19: Bandwidth for big block sizes - Hopper 48 cores

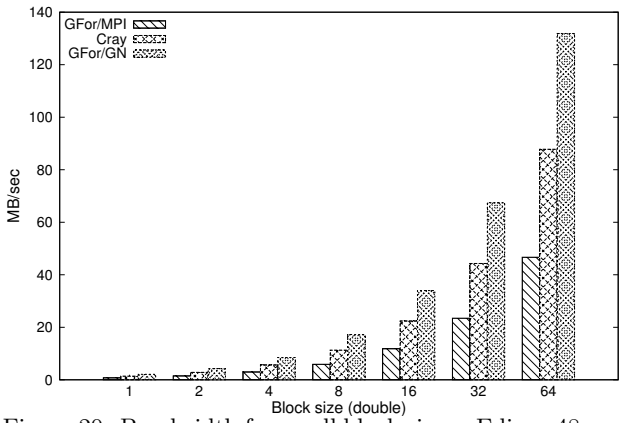

Figure 20: Bandwidth for small block sizes - Edison 48 cores

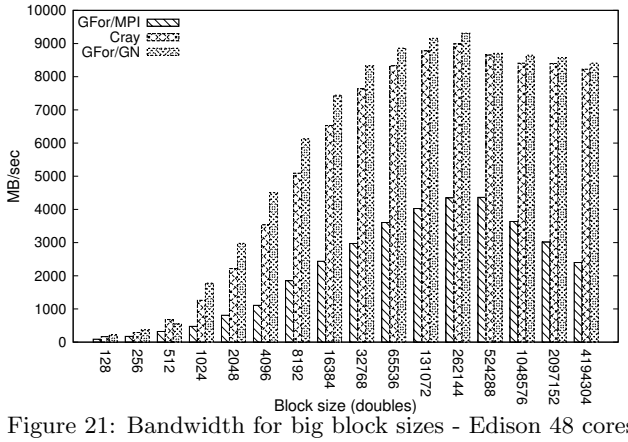

Figure 21: Bandwidth for big block sizes - Edison 48 cores

5.2.6 *Single pt2pt Strided Get on multiple nodes*

The strided transfers on multiple node, for both Hopper and Edison, have a unexpected trend. Figures 24 and 25 show that LIBCAF_MPI has better performance than Cray on both machines.

5.3 Burgers Solver – GFortran vs. Intel

The Burgers Solver brings us closest to a complete scientific application. It uses coarrays mainly for scalar transfers between neighbor images. In other words, the scalar transfers usually occur within the same node. Figure 26 shows that,

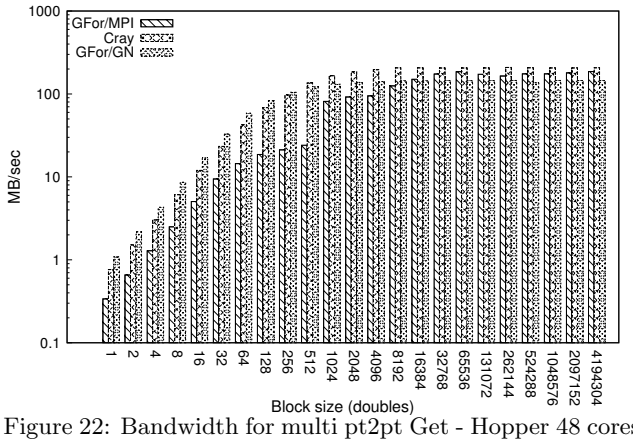

Figure 22: Bandwidth for multi pt2pt Get - Hopper 48 cores

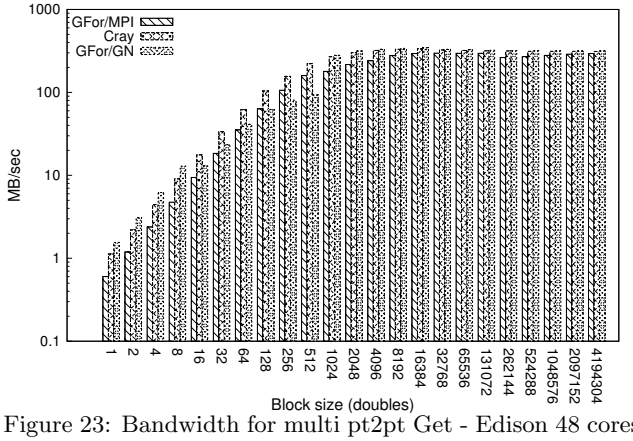

Figure 23: Bandwidth for multi pt2pt Get - Edison 48 cores

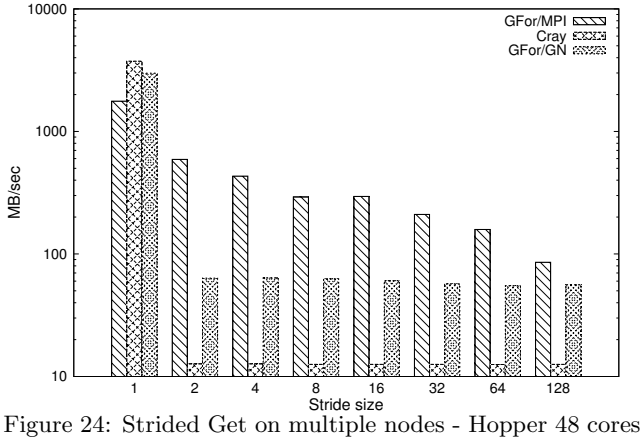

Figure 24: Strided Get on multiple nodes - Hopper 48 cores

on 16 cores (one compute node) Intel outperforms GFortran (as stated in 5.1.1 and 5.1.2). On multiple compute nodes, GFortran slightly outperforms Intel. The small difference stems from the communication being between neighboring images that are usually on the same node.

5.4 Burgers Solver – GFortran vs. Cray

Figure 27 shows Burgers Solver performance on Hopper. Cray outperforms GFortran – probably owing to leveraging Cray's proprietary communication library running on Cray's proprietary interconnect.

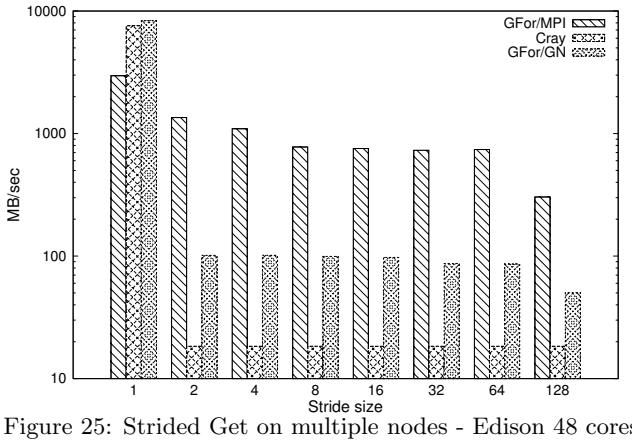

Figure 25: Strided Get on multiple nodes - Edison 48 cores

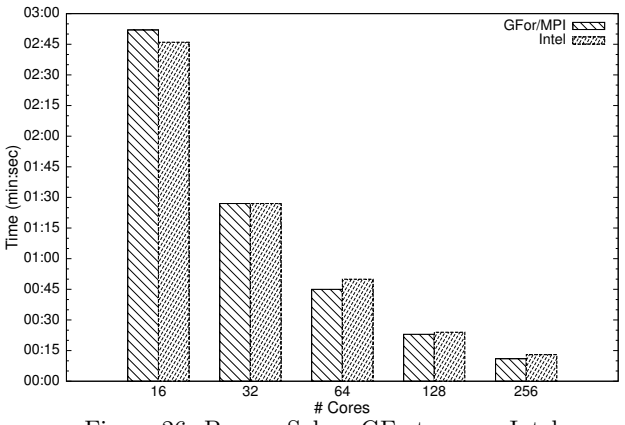

Figure 26: BurgersSolver GFortran vs. Intel

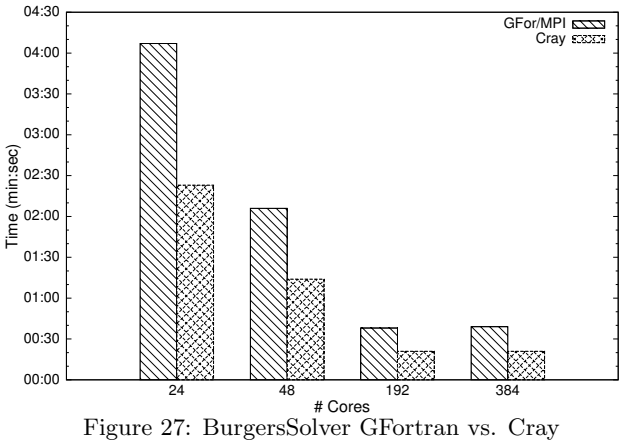

Figure 27: BurgersSolver GFortran vs. Cray

5.5 CAF Himeno - GFortran vs. Intel

CAF Himeno uses the Jacobi method for a 3D Poisson relaxation. The 3D nature of the problem implies strided transfers among several images. Using 64 cores (4 Yellowstone nodes), Intel requires more than 30 minutes to complete. We therefore report in Figure 28 only the results for 16 and 32 cores.

In this case, we report the MFLOPS on the y axis, thus higher means better.

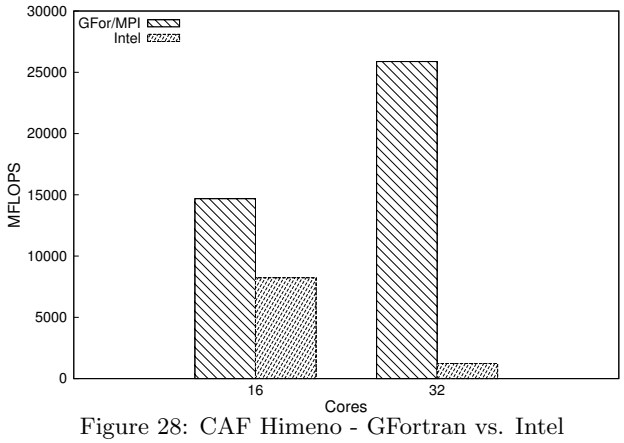

Figure 28: CAF Himeno - GFortran vs. Intel

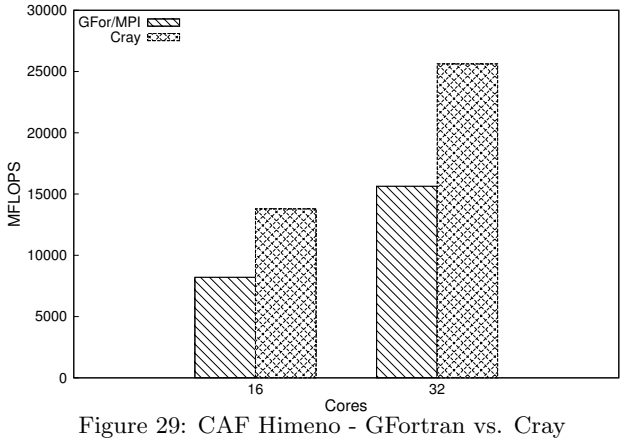

Figure 29: CAF Himeno - GFortran vs. Cray

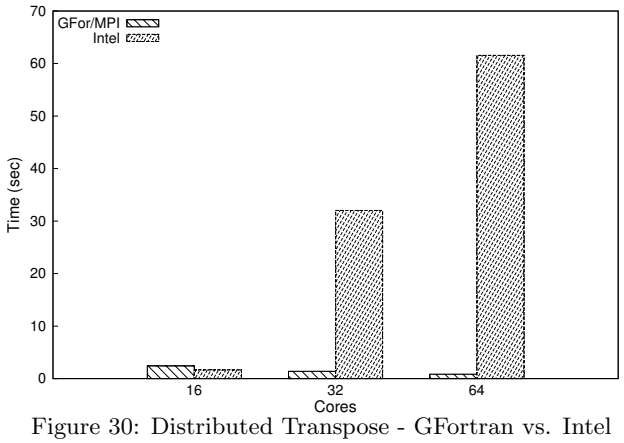

Figure 30: Distributed Transpose - GFortran vs. Intel

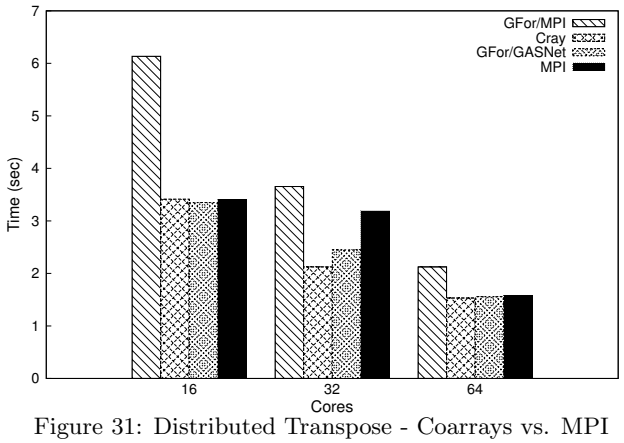

Figure 31: Distributed Transpose - Coarrays vs. MPI

5.6 CAF Himeno: GFortran vs. Cray

The execution of CAF Himeno on Hopper required quite a bit of tuning. In order to run the 32-core test, we were forced to place 8 images on each node. Because each node has 24 cores, we wasted 16 cores on each node for memory reasons. Gfortran proved the easiest coarray implementation for CAF Himeno. Figure 29 shows the CAF Himeno results for Cray, which outperforms GFortran on single and multiple nodes.

5.7 Distributed Transpose: GFortran vs. Intel

This test case performs transpose operations that involve sending a contiguous, four-dimensional array. Figure 30 shows again that Intel outperforms GFortran on a single node.

On multiple nodes, however, the time required by Intel explodes because the communication is spread over several processes among the nodes (and not only between images on the same node). Inspection of the assembly language code generated Intel shows element-by-element transfers with lock-/unlock pairs bracketing each transfer.

5.8 Distributed Transpose – GFortran vs. Cray

For this case only, we provide a comparison between coarrays and MPI. We used only 16 of the 24 cores provided by Hopper because the matrix size must be a multiple of the number of processes involved. We fixed the size at 1024x1024x512 elements. Figure 31 shows that, within the same node (16 cores), GFortran with LIBCAF_GASNet has the best performance, even better than Cray. On multiple nodes, Cray shows the best performance. Notably, GFortran with LIBCAF_GASNet outperforms a pure MPI implementation in every configuration.

6. CONCLUSIONS AND FUTURE WORK

In this section we summarize the conclusions of our investigation and our plans for further development.

6.1 GFortran vs. Intel

Our tests showed Intel outperforming GFortran only during small transfers within the same node. The bad performance showed by Intel is mainly related with an element-by-element transfer, even with contiguous memory segments (see Figure 7).

Since Intel sends one element at time, its superior performance for scalar transfers most likely relates to good performance provided by IntelMPI. On multiple nodes, the penalty of sending element-by-element becomes huge for Intel.

GFortran shows better performance than Intel on array transfers within the same node and in every configuration involving the network.

6.2 GFortran vs. Cray

On Hopper, GFortran outperformed Cray for small transfers on a single node, while Cray outperformed GFortran in most cases for big transfers. On Edison, GFortran outperforms Cray in both single- and multiple-node configurations. An unexpected result for us is the poor performance of Cray during strided transfers on multiple nodes. Also, although Cray proposes a complete and efficient coarray implementation, it still requires some tuning in order to run the programs.

Since Cray does not provide the source code of its coarray implementation, we are not able to explain why the strided transfer between multiple nodes has very poor performance. LIBCAF_MPI uses the MPI Derived Data Types (MPI_Vector for 1-D arrays and MPI_Type_Indexed for any other case) in order to implement the efficient strided transfer support. This approach produces good performance but requires a lot of memory.

6.3 Final Considerations

Although LIBCAF_GASNet often performs better than LIBCAF_MPI, the latter provides a more easy-to-use support for coarrays. The biggest limitation of LIBCAF_GASNet consists in the necessity to declare the total amount of memory to use for coarrays. Sometimes this quantity could not be known a-priori and several attempts are needed in order to find a good estimation.

Our intent, in this work, is to provide free, stable, easy-to-use and efficient support for coarrays. LIBCAF_MPI enables GFortran to provide coarray support that works on any architecture able to compile GCC and MPI (a very common configuration). We expect GFortran/LIBCAF_MPI to work on Linux, OS X, and Windows without any limitations at no cost and with strong performance. LIBCAF_GASNet, even if slightly more hard to use, provides great portability (some configurations require just a C89 compiler and MPI-1) and great performance.

GFortran's multi-image coarray support is already on GCC 5.0 trunk freely available for download and installation. The http://opencoarrays.org web site includes a link to a git repository that offers public, read-only access to the full OpenCoarrays library (which includes LIBCAF_MPI and LIBCAF_GASNet).

6.4 Future Work

We plan to improve our strided transfer support and to cover all the missing features from the standard. Because LIBCAF_GASNet has shown remarkable performance in every configuration tested, we also plan to improve that version in order to cover missing features and to improve the usage and installation process. Finally, we plan to make a complete performance comparison of MPI and GASNet with other communication libraries like ARMCI [9] and OpenSHMEM [3].

Acknowledgments

We gratefully acknowledge the support we received from the following institutions: National Center for Atmospheric Research for the access on Yellowstone/Caldera and the logistic support provided during the development of this work.
CINECA for the access on Eurora/PLX for the project HyPS-BLAS under the ISCRA grant program for 2014.
Google, because part of this work is a Google Summer of Code 2014 project.
National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, for the access on Hopper/Edison under the grant OpenCoarrays.

7. REFERENCES

[1] D. Bonachea. GASNet Specification, v. 1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.

[2] R. Chandra. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[3] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proc. of 4th Conf. on Partitioned Global Address Space Programming Model*, PGAS '10. ACM, 2010.

[4] I. D. Chivers and J. Sleightholme. Compiler support for the Fortran 2003 and 2008 standards. *ACM SIGPLAN Fortran Forum*, 33(2), Aug. 2014.

[5] D. Eachempati, H. J. Jun, and B. Chapman. An open-source compiler and runtime implementation for Coarray Fortran. In *Proc. of 4th Conf. on Partitioned Global Address Space Programming Model*, PGAS '10. ACM, 2010.

[6] G. Jin, J. Mellor-Crummey, L. Adhianto, W. N. Scherer III, and C. Yang. Implementation and performance evaluation of the HPC challenge benchmarks in Coarray Fortran 2.0. In *Proc. of IEEE Int'l Parallel and Distributed Processing Symposium*, IPDPS '11, pages 1089–1100, 2011.

[7] M. Haveraaen, K. Morris, D. W. I. Rouson, H. Radhakrishnan, and C. Carson. High-performance design patterns for modern Fortran. *Scientific Programming*, in press.

[8] D. Henty. A parallel benchmark suite for Fortran Coarrays. In *Applications, Tools and Techniques on the Road to Exascale Computing*, pages 281–288. IOS Press, 2012.

[9] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Parallel and Distributed Processing*, volume 1586 of *LNCS*, pages 533–546. Springer Berlin Heidelberg, 1999.

[10] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[11] R. W. Numrich and J. Reid. Co-arrays in the next Fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, Aug. 2005.

[12] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[13] R. S. Rogallo. Numerical experiments in homogeneous turbulence. Technical Report 81315, National Aeronautics and Space Administration, 1981.

[14] D. Rouson, J. Xia, and X. Xu. *Scientific Software Design: The Object-Oriented Way*. Cambridge University Press, New York, NY, 2011.

[15] A. Vaught. The G95 project. http://g95.org, Dec. 2008.